O'REILLY

TURING 图灵程序设计丛书



学习敏捷 构建高效团队

Learning Agile

精讲精益、Scrum、极限编程和看板方法，全面解读敏捷价值观及原则，
提高团队战斗力

[美] Andrew Stellman Jennifer Greene 著
段志岩 郑思遥 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

学习敏捷：构建高效团队

Learning Agile

[美] Andrew Stellman，Jennifer Greene 著
段志岩 郑思遥 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　　京

图书在版编目（CIP）数据

学习敏捷 : 构建高效团队 / (美) 安德鲁·斯特尔曼 (Andrew Stellman) , (美) 珍妮弗·格林 (Jennifer Greene) 著 ; 段志岩, 郑思遥译. -- 北京 : 人民邮电出版社, 2017.2
（图灵程序设计丛书）
ISBN 978-7-115-44755-5

Ⅰ. ①学… Ⅱ. ①安… ②珍… ③段… ④郑… Ⅲ. ①软件开发—研究 Ⅳ. ①TP311.52

中国版本图书馆CIP数据核字(2017)第018932号

内 容 提 要

本书以敏捷软件开发为中心，系统阐述了敏捷原则和实践的先进理念和重要意义，并分别讲解了 Scrum、极限编程、精益和看板四套敏捷实践的应用。作者从开发团队的日常困境入手，用讲故事的形式展开问题，由表及里，层层讲解，并在每章最后附上参考图书，便于读者进一步查找学习。本书内容生动，语言通俗易懂，集趣味性和实用性于一体，是学习敏捷开发、提升团队效率的极佳参考书。

本书适合软件开发人员、项目经理、软件项目主管阅读。

◆ 著　[美] Andrew Stellman，Jennifer Greene
译　段志岩　郑思遥
责任编辑　朱　巍
执行编辑　张　憬　赵瑞琳
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：19.25
字数：456千字　2017年2月第1版
印数：1－3 500册　2017年2月北京第1次印刷

著作权合同登记号　图字：01-2015-5422号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

谨以此书献给 Nisha 和 Lisa，感谢她们一直以来对我们的耐心。

本书赞誉

又一本来自 Andrew 和 Jennifer 团队的佳作。他们的文字风格引人入胜，他们对敏捷知识的了解深刻透彻。他们的书不仅内容全面，而且很有实践意义。

——Grady Booch，IBM 院士

对于高效敏捷团队的构建，最大的障碍不在于学习敏捷方式，而在于理解敏捷。要释放团队潜力，让成员全力以赴，并且在合作中寻求创新，关键在于帮助他们了解团队为什么要敏捷。Andrew 和 Jennifer 关注价值观和原则，以极佳的方式帮助你和你的团队发掘敏捷背后的缘由。我等不及要和大家分享这本书了。

——Todd Webb，国际电子商务公司的技术产品主管

我很热衷于敏捷，而《学习敏捷》教会了我如何在自己的组织中推广敏捷。这本书以引人入胜的方式让你深入了解敏捷原则和实践。这里的故事很有代入感，你很容易借此说服团队成员转向敏捷，然后享受成果。

——Mark Denovich，Centriq Group 公司高级业务顾问兼美国开发部主管

这是一本非常棒的指导书，适合任何想要深刻理解敏捷的团队成员。Stellman 和 Greene 以清楚明了而又引人入胜的笔触讲解了敏捷价值观和原则。这本书中的幽默、例子以及精道的比喻让人耳目一新。不过这本书最大的优点还是明确指出了敏捷团队时常遇到的问题，并且提供了取得进展、获得更多成果的实用建议。

——Matthew Dundas，Katori 公司 CTO

作为一名工程师，我一直认为敏捷实践解决的是行业中的大问题。其实做到敏捷很难，这不仅仅是实践的问题。正如作者所说，零碎的敏捷方法只能带来“聊胜于无”的结果。如果你刚刚开始实践敏捷，或者只做到了“聊胜于无”，那么你可以从这本书中得到很多实用建议，Andrew 和 Jennifer 会告诉你如何深入理解“敏捷宣言”，真正做到敏捷。

——James W Grenning，Wingman 软件公司创始人，敏捷宣言起草人之一

Andrew Stellman 和 Jennifer Greene 的这本书写得非常棒。书中全面整合了很多实战资源，想要学习敏捷的人都可以轻松获取。他们在书中谈到了敏捷的诸多方面，而不只是讲解了敏捷团队的理想状态。通过挖掘敏捷的不同要素，他们呈现了敏捷的标准实践和最佳效果，也指出了人们对敏捷的常见误解及对应的结果。此外还讲解了具体的实践和做法会对不同职责的个体造成怎样的影响。对于初学者和有经验的敏捷实践者而言，这本书都是很好的学习资源。

——Dave Prior，敏捷顾问兼教练

要想学习敏捷的具体方法，你需要阅读相关的书籍。这就意味着你要事先想好自己需要什么。你真有这么敏捷吗？ Andrew 和 Jenny 提供了一套关于敏捷的纲要，实用易懂，可以让你正确理解敏捷的概念。你不用提前确定自己需要什么样的敏捷方法，看看这本书就可以做决定了。你可以通过这本书了解敏捷的体系，以及它是如何运作的。

——Johanna Rothman，作家兼顾问

在软件开发团队中，相比专业知识和工具，文化氛围对于项目的成功更为重要。关于如何将不同人的割裂视角凝聚成全体成员的统一视角，让团队共享价值观和实践，Stellman 和 Greene 的建议能够为任何组织的项目经理提供帮助。他们比较了 Scrum、极限编程、精益和看板方法，分析了敏捷原则的多种实践方式。书中生动的例子解释了人们在走向敏捷的过程中所遇到的困境以及收获的成果。

——Patricia Ensworth，Harborlight 管理服务有限责任公司董事长

《学习敏捷》这本书很全面、很易懂、很实用、很有趣。书中讲解的价值观、原则和方法非常有启发，我等不及要在团队中实践了。

——Sam Kass，软件架构师及财务部门的技术负责人

这是一本非常好的书，可以为任何层次的软件专业人士介绍敏捷方法。它可以帮助你了解进而避免开发团队遇到的常见陷阱。

——Adam Reeve，大型社交网站的工程师和团队主管

目录

序 xv

前言 xvii

第 1 章　学习敏捷 1

1.1　什么是敏捷 2
1.2　本书的读者对象 5
1.3　本书的目标 6
1.4　努力建立敏捷思维 6
1.5　本书结构 9

第 2 章　理解敏捷价值观 11

2.1　团队主管、架构师和项目经理走进了一间酒吧…… 12
2.2　没有银弹 14
2.3　敏捷可以拯救乱局吗 16
2.3.1　引入敏捷，带来变化 17
2.3.2　“聊胜于无”的结果 18
2.4　视角割裂 19
2.4.1　视角割裂带来的问题 21
2.4.2　为什么视角割裂只能做到“聊胜于无” 22
2.5　敏捷宣言帮助团队认识实践的目的 24
2.5.1　个体和互动高于流程和工具 25
2.5.2　可工作的软件高于详尽的文档 25
2.5.3　客户协作高于合同谈判 26
2.5.4　响应变化高于遵循计划 26

2.5.5 原则高于实践 ······27
2.6 理解敏捷的“大象”······28
2.7 着手采用一套新方法······32

第 3 章 敏捷原则······37

3.1 敏捷软件开发的 12 条原则······38
3.2 客户总是对的吗······38
3.3 交付项目······40
3.3.1 原则 1：最优先要做的是尽早、持续地交付有价值的软件，让客户满意 ······40
3.3.2 原则 2：欣然面对需求变化，即使是在开发后期。敏捷过程利用变化为客户维持竞争优势 ······41
3.3.3 原则 3：频繁交付可工作的软件，从数周到数月，交付周期越短越好 ······42
3.3.4 改进电子书阅读器团队的项目交付计划 ······44
3.4 沟通和合作······46
3.4.1 原则 4：在团队内外，面对面交谈是最有效、也是最高效的沟通方式 ······48
3.4.2 原则 5：在整个项目过程中，业务人员和开发人员必须每天都在一起工作 ······49
3.4.3 原则 6：以受激励的个体为核心构建项目，为他们提供环境和支持，相信他们可以把工作做好 ······51
3.4.4 在电子书阅读器项目中采用更好的沟通方式 ······52
3.5 项目实施——推进项目······53
3.5.1 原则 7：可工作的软件是衡量进度的首要标准 ······53
3.5.2 原则 8：敏捷过程倡导可持续开发。赞助商、开发人员和用户要能够共同、长期维持其步调，稳定向前 ······54
3.5.3 原则 9：坚持不懈地追求技术卓越和设计优越，以此增强敏捷的能力 ······55
3.5.4 改善电子书阅读器团队的工作环境 ······55
3.6 项目和团队的持续改进······56
3.6.1 原则 10：简单是尽最大可能减少不必要工作的艺术，是敏捷的根本 ······56
3.6.2 原则 11：最好的架构、需求和设计来自自组织的团队······57
3.6.3 原则 12：团队定期反思如何提升效率，并依此调整 ······57
3.7 敏捷项目：整合所有原则······58

第 4 章 Scrum 和自组织团队······62

4.1 Scrum 的规则 ······64
4.2 第 1 幕：Scrum 的适用条件 ······65
4.3 Scrum 团队中每个人都要对项目负责 ······67
4.3.1 Scrum 主管指导团队的决策······67
4.3.2 产品所有者帮助团队了解软件的价值 ······68
4.3.3 每个人都对项目负责 ······69
4.3.4 Scrum 有一组自己的价值观······75

4.4 第 2 幕：状态更新只是社交网络的玩法……78
4.5 整个团队参与每日 Scrum 例会……80
4.5.1 反馈和“可见－检查－调整”周期……80
4.5.2 最后责任时刻……81
4.5.3 召开有效的每日 Scrum 例会……83
4.6 第 3 幕：将冲刺计划写到墙上……86
4.7 冲刺、计划和回顾会议……87
4.7.1 迭代式与增量式……87
4.7.2 冲刺成也在于产品所有者，败也在于产品所有者……89
4.7.3 可见性和价值观……89
4.7.4 计划并执行有效的 Scrum 冲刺……93
4.8 第 4 幕：尽力之后……94

第 5 章 Scrum 计划和集体承诺……99

5.1 第 5 幕：出乎意料……100
5.2 用户故事、速度和普遍接受的 Scrum 实践……102
5.2.1 提升软件价值……102
5.2.2 以用户故事构建用户真正会用到的功能……103
5.2.3 满意条件……105
5.2.4 故事点和速度……106
5.2.5 燃尽图……108
5.2.6 通过用户故事、故事点、任务和任务板来计划并实施冲刺……111
5.2.7 广受认可的 Scrum 实践……115
5.3 第 6 幕：第一次胜利……116
5.4 回顾 Scrum 价值观……116
5.4.1 具体实践没有价值观也有效果（只是别管它叫 Scrum）……117
5.4.2 你的公司文化与 Scrum 的价值观兼容吗……119

第 6 章 极限编程与拥抱变化……128

6.1 第 1 幕：开始加班……129
6.2 极限编程的主要实践……130
6.2.1 编程实践……130
6.2.2 集成实践……131
6.2.3 计划实践……132
6.2.4 团队实践……133
6.2.5 为什么开发团队抵制变化，上述实践如何提供帮助……134
6.3 第 2 幕：计划有变，但我们还是看不到希望……137
6.4 极限编程的价值观帮助团队改变心态……139
6.4.1 极限编程帮助开发人员学会与用户协作……141

6.4.2 开发团队的怀疑会破坏实践的效用 …… 142
6.5 正确的思维从极限编程的价值观开始 …… 144
6.5.1 极限编程的价值观 …… 144
6.5.2 以善意铺就 …… 144
6.6 第 3 幕：势头的变换 …… 147
6.7 理解极限编程价值观，拥抱变化 …… 148
6.7.1 极限编程的指导原则 …… 149
6.7.2 极限编程指导原则可以加深对计划的理解 …… 151
6.7.3 极限编程指导原则与实践相互促进 …… 152
6.7.4 反馈循环 …… 154

第 7 章 极限编程、简化和增量式设计 …… 163

7.1 第 4 幕：再次加班 …… 164
7.2 代码和设计 …… 165
7.2.1 代码异味和反模式（如何判断你是不是聪明过头了）…… 166
7.2.2 极限编程团队主动寻找和修复代码异味 …… 168
7.2.3 钩子、边界情况以及功能过多的代码 …… 170
7.2.4 代码异味会增加复杂性 …… 175
7.3 把编码和设计决定留到最后责任时刻 …… 175
7.3.1 决然重构，偿还技术债务 …… 177
7.3.2 持续集成，排查设计问题 …… 179
7.3.3 避免一体式设计 …… 180
7.4 增量式设计与极限编程的整体实践 …… 182
7.4.1 有时间进行思考，团队才能做好工作 …… 184
7.4.2 团队成员彼此信任并共同作出决定 …… 186
7.4.3 极限编程的设计、计划、团队和整体实践形成了一个带动创新的系统 …… 186
7.4.4 增量式设计与为了复用而设计 …… 188
7.4.5 简化单元交互，系统实现增量式成长 …… 190
7.4.6 优秀的设计源自简单的交互 …… 190
7.5 第 5 幕：最终得分 …… 192

第 8 章 精益、消除浪费和着眼全局 …… 200

8.1 精益思维 …… 201
8.1.1 你已经理解了很多精益价值观 …… 201
8.1.2 承诺、选择意识和集合式开发 …… 203
8.2 第 1 幕：还有一件事…… …… 207
8.3 创造英雄与神奇思维 …… 209
8.4 消除浪费 …… 210
8.5 加深对产品的理解 …… 214

8.5.1 着眼全局 ······216
8.5.2 找到问题的根本原因 ······218
8.6 尽快交付 ······219
8.6.1 使用面积图可视化工作进度 ······221
8.6.2 限制进行中的工作，控制瓶颈 ······225
8.6.3 拉动式系统帮助团队消除约束 ······226

第 9 章 看板方法、流程和持续改进 ······233

9.1 第 2 幕：紧赶慢赶的游戏 ······234
9.2 看板方法的原则 ······236
9.2.1 找到一个出发点并由此进行实验性的演进 ······236
9.2.2 用户故事进去，代码出来 ······238
9.3 用看板方法改进流程 ······240
9.3.1 将工作流程可视化 ······241
9.3.2 限制进行中的工作 ······246
9.4 测量并管理流量 ······251
9.4.1 用 CFD 和进行中工作面积图测量并管理流量 ······252
9.4.2 用利特尔法则控制系统的流量 ······259
9.4.3 用进行中工作上限管理流量，自然地创造缓冲 ······263
9.4.4 让过程策略明确统一 ······265
9.5 看板方法下自然发生的行为 ······266

第 10 章 敏捷教练 ······275

10.1 第 3 幕：还有一件事（又来了?！） ······276
10.2 教练要理解人们为什么不想改变 ······277
10.3 教练要理解人们如何学习 ······280
10.4 教练清楚如何让一套方法起作用 ······284
10.5 进行敏捷指导时的原则 ······285

关于作者 ······288

关于封面 ······288

序

人们似乎总是需要辩论点什么。Van Halen 跟 David Lee Roth 在一起更好，还是跟 Sammy Hagar 在一起更好？百事可乐更好喝，还是可口可乐更好喝？Lennon 唱得好，还是 McCartney 唱得好？养猫还是养狗？早期，敏捷方法也有原则与实践之争。早期的敏捷倡导者就一组原则达成了共识，并将其庄严载入敏捷宣言（Agile Manifesto），而多个敏捷方法之间也共享了很多具体实践。然而，当时人们就一个问题展开了激烈的争论，即一个团队是应该先理解敏捷软件开发的原则，还是应该先开始具体实践。

支持从实践开始的人认为熟能生巧。按照敏捷的方式行事，团队就会变得敏捷。通过采用敏捷方法的实践，如结对编程、测试和构建自动化、使用迭代、与主要的利益干系人紧密合作等，团队就会逐渐理解敏捷原则。

而支持从原则开始的人则认为没有原则支撑的实践是空洞的。在不明就里的情况下采用敏捷实践不会带来敏捷。敏捷一直关注持续改进。这些人认为，如果不理解自己所做的事情，团队就没法做到持续改进。

在《学习敏捷》这本书中，Andrew Stellman 和 Jennifer Greene 让原则和实践并重。在这一点上，他们是我所见过的人中做得最好的。他们指出，在不了解敏捷的情况下，实践只能带来所谓“聊胜于无”的成功。也就是说，只采用敏捷的实践是有帮助的，但是与真正的敏捷所带来的成功相比，还差得很远。

我第一次与 Andrew 和 Jennifer 见面是在六年前。当时他们为自己的新书《团队之美》（*Beautiful Teams*）来采访我。尽管那本书的书名中并没有“agile”（敏捷）这样的字眼，但是从很多方面看，那就是一本关于敏捷的书。拥抱敏捷原则、掌握所需敏捷实践并且摒弃不必要敏捷实践的团队确实是出色的团队。在《学习敏捷》这本书中，Andrew 和 Jennifer 集中讨论了当今三种最常见的敏捷方法：Scrum、极限编程和看板方法，并将讨论聚焦于敏捷这一主题。读者将看到基于共有原则的这三种方法如何形成不同的实践。比如，如果想知道为什么 Scrum 需要在冲刺结束时进行回顾而极限编程却不需要，在这里就能找到答案。

通过与 Andrew 和 Jennifer 一起探索 Scrum、极限编程、精益和看板方法，读者将读到很多故事。这样安排是有道理的。毕竟，很多敏捷团队的一个共同实践就是通过用户故事来描述某个系统的用户想要什么。读者将看到这样一些团队，他们费尽周折想要开发出正确的

软件功能，为交付去年的需求花费了太长时间，错把敏捷当成另一种形式的上令下行式管理方法，不去拥抱变化而是被变化折磨得死去活来，等等。更重要的是，本书将向读者介绍团队克服这些问题的方法，让读者学以致用。

《学习敏捷》这本书彻底结束了“原则和实践哪一个应该在先”的争论。书中引人入胜的故事和鞭辟入里的评论说明了一个简单的真理：在敏捷中原则与实践是不可分割的。通过阅读本书，读者将对如何成为真正出色的团队（或回到成为真正出色团队的正轨上来）有更为深入的理解。

Mike Cohn

《Scrum 敏捷软件开发》《用户故事与敏捷方法》作者

博尔德，美国科罗拉多州

前言

致谢

我们写作本书的目的是帮助读者学习敏捷。本书的完成离不开大家的帮助和支持。首先，我们要感谢优秀的编辑 Mary Treseler。从我们初次在曼哈顿市中心的一家印度餐厅跟她讨论本书的那天开始，一直到今天我们看到这本书，在这过程中，她付出了极大的努力。她在我们与 O'Reilly 出版社的合作中起到了十分重要的作用。没有她的支持，我们就不会有今天的成绩。

我们也要感谢 O'Reilly 出版社的其他工作人员，没有他们就不可能有这本书。他们是 Mike Hendrickson、Laurie Petrycki、Tim O'Reilly、Ally MacDonald、Andy Oram 和 Nicole Shelby，我们尤其要感谢 Marsee Henon、Sara Peyton，以及在塞巴斯托波的所有杰出的媒体人员和公关人员。

我们感谢 Mike Cohn 为本书撰写的精彩序言，同时感谢他多年来给予我们的极好建议。我们还想感谢他撰写了那么多优秀的图书，我们从中真的学到了很多东西！我们还要感谢 David Anderson，针对本书的第 8 章和第 9 章给出了非常好的反馈意见。我们要感谢 Grady Booch、Scott Ambler、James Grenning、Scott Berkun、Steve McConnell、Karl Wiegers、Johanna Rothman、Patrica Ensworth、Tommy Tarka、Keoki Andrus、Neil Siegel、Karl Fogel 和 Auke Jilderda。他们多年来为我们提供了极好的素材，尤其是为《团队之美》一书。而且我们要特别感谢 Barry Boehm，他不仅为《团队之美》贡献了一个非常精彩的故事，更重要的是奠定了敏捷的智力基础。我们还要感谢 Kent Beck、Alistair Cockburn、Ken Schwaber、Jeff Sutherland、Ron Jeffries、Tom Poppendieck、Mary Poppendieck、Lyssa Adkins 和 Jim Highsmith。他们在敏捷领域做出了突破性的工作。可以说，没有他们，我们不可能写出此书。

我们还要感谢所有的技术审阅人：Faisal Jawdat、Adam Reeve、Anjanette Randolph、Samuel Weiler、Dave Prior、Randy DeFauw、Todd Webb、Mickael DeWitt 和 Paul Ellarby。他们的反馈十分到位，审校非常细致。

最后，我们要感谢数百位软件团队成员，感谢他们不吝与我们分享多年来的问题、解决方案、故事以及经历。

Andrew 要感谢 Lisa Kellner。他同时也要感谢卡内基梅隆大学计算机科学系所有启迪过他的人，尤其是 Bob Harper、Mark Stehlik 和 Randy Bryant。他要感谢多年的良师益友 Tony Visconti。他要感谢他的朋友 Sara Landeau、Greg Gassman、Juline Koken、Kristeen Young 和 Casey Dunmore。关于团队合作，他从这些优秀的音乐家身上学到了很多。回头想想，这是多么不可思议。他还要感谢职业生涯中共事过的优秀同事，包括 Dan Faltyn、Ned Robinson、Debra Herschmann、Mary Gensheimer、Lana Sze、Warren Pearson、Bill DiPierre、Jonathan Weinberg 和 Irene O'Brien。最后还要感谢曾工作过的两个最佳软件团队的同事，他们是 Optiron 的 Mark Denovich、Eric Renkey 和 Chris Winters，以及美国银行的 Mike Hickin、Nick Lai、Sunanda Bera 和 Rituraj Deb Nath。

Jennifer 要感谢 Nisha Sondhe。她还要感谢 Christopher Wenger、Brian Romeo、LaToya Jordan、Mazz Swift、Rekha Malhotra、Courtney Nelson、Anjanette Randolph、Shona McCarthy、Ethan Hill、Yeidy Rodriguez、Kyle Mosier、Achinta McDaniel、Jaikaran Sawhny 和 Kit Cole，感谢他们的支持和陪伴。她要感谢家人在她写作本书的将近三年中给予她的耐心和鼓励。她要感谢 Tanya Desai 和 Dilan Desai 夫妇的支持和帮助。最后，她要感谢许多优秀的同事。多年来，她从他们身上学到了很多东西。还需要感谢的人太多，无法一一列举，这里只列出一小部分，他们是 Joe Madia、Paul Oakes、Jonathan Weinberg、Bianka Buschbeck、Thor List、Oleg Fishel、Brian Duperrouzel、Dave Murdock、Flora Chen、Danny Wunder、David San Filippo 和 Rasko Ristic。

Safari® Books Online

Safari Books Online（http://my.safaribooksonline.com/?portal=oreilly）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品（http://www.safaribooksonline.com/content）。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

Safari Books Online 为组织（http://www.safaribooksonline.com/organizations-teams）、政府机构（http://www.safaribooksonline.com/government）和个人读者（http://www.safaribooksonline.com/individuals）提供了一系列的产品组合（http://www.safaribooksonline.com/subscriptions）和价格体系。订阅者可在一个支持完全搜索的数据库中访问数以千计的图书、培训视频和尚未发行的书稿。发行这些内容的是 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他数十家发行商（http://www.safaribooksonline.com/publishers）。要想了解 Safari Books Online 的更多信息，请访问我们的网站（http://www.safaribooksonline.com/）。

第 1 章

学习敏捷

> 对继续学习的渴望是一个人可以形成的最重要的态度。
>
> ——约翰 · 杜威，《经验与教育》

这是一个令人激动的敏捷时代。对于一直困扰软件开发团队的种种问题，IT 行业首次找到了真正可持续的解决方案。以下是敏捷承诺的几种解决方案。

- 敏捷项目可以按时完成，为那些苦恼于交付期限和预算的团队带去了福音。
- 敏捷项目交付高质量的软件，那些受困于 bug 和低效软件的团队会迎来巨大的变革。
- 敏捷团队构建的代码结构优良且易于维护，那些常常维护复杂又混乱的代码的团队会得到解脱。
- 敏捷团队会让用户满意，不再交付无法为用户带来价值的软件。
- 最重要的是，在出色的敏捷团队中，开发人员不用加班，可以与亲朋好友共度晚间时光和周末。这在他们的职业生涯中可能是史无前例的。

敏捷之所以流行，是因为很多走敏捷路线的团队都表示收获颇丰。它们构建出了更好的软件，团队成员间合作更为愉快，用户的需求也得到了更好的满足，工作环境也更加轻松愉悦。一些团队在接纳敏捷理念之后终于可以在困扰多年的问题上有所进展。那么，伟大的团队如何利用敏捷理念构建更好的软件？更准确地说，我们自己怎样才能利用敏捷理念取得如此好的结果呢？

在本书中，读者会学到两种最流行的敏捷方法：Scrum 和极限编程（eXtreme Programming，XP）。此外，读者还会学习到精益（Lean）方法和看板（Kanban）方法，了解如何通过它们理解当前采用的软件构建方法，并改进当前的状况。通过本书，读者还将明白一点：尽管这四种敏捷学派关注的是软件开发中的不同领域，但它们有一个重要的共同特点：重视改变团队的思维模式。

正是因为转变了思维模式，一个仅仅接触敏捷实践皮毛的团队就可以获得提升，从而真正改进其软件构建方式。本书的目标是帮助读者深入了解敏捷的两个方面：一方面是日常工作中的种种实践，另一方面是敏捷的价值观和原则。后者能够帮助团队从根本上改变构建软件的思维模式。

1.1 什么是敏捷

敏捷是指能够让团队思考更加有效、工作更为高效，并且作出更好决策的一*组方法和相关理念*。

这些方法和理念适用于传统软件工程的所有领域，例如项目管理、软件设计、软件架构和流程改进。每种方法和理念都包括*实践*。这些实践经过了精简和优化，应用起来非常方便。

敏捷也是一种*思维模式*，思维模式的正确与否会影响团队具体实践的高效程度。正确的思维模式有助于团队成员共享信息，从而共同作出重要的项目决策，而不是让经理独自负责所有的决策。敏捷思维模式涉及整个团队的开工计划、设计和流程改进。在敏捷团队所采用的实践方式中，大家不仅共享信息，而且对实践的应用方式都有发言权。

敏捷在有些团队中还没有取得成功，这造成了现实和承诺之间的差距。这种差距形成的关键往往在于项目团队的思维模式。很多构建软件的公司都尝试过敏捷开发。许多团队尝到了成功的甜头，而一些团队却结果平平。尽管这些团队在项目运作上有所改善，这些改善也足以证明采用敏捷的努力是值得的，但是他们并没有看到敏捷方法承诺的那些实质性改变。这就是改变思维模式的意义所在。走敏捷路线意味着帮助团队找到一种高效的思维模式。

那么，改变思维模式到底意味着什么？如果你是软件团队中的一员，那么你每天的工作就是规划、设计、构建和发布软件。思维模式与这些有什么关系呢？事实证明，你在日常工作中采取的实践很大程度上取决于你和其他团队成员对待实践的态度。

举个例子：*每日站立会议*（daily standup）是很多团队都采用的一种最常见的敏捷实践。在每日站立会议上，每一位成员都会讲述自己手头的工作以及面临的挑战。为了使会议简短，大家在会议期间全程站立。很多团队在项目中引入每日站立会议，因而获得了很大成功。

假设有一位刚开始学习敏捷的项目经理，他想在项目中引入每日站立会议。令他意外的是，并不是所有人都像他那样对这一新实践欣喜若狂。组里的一位开发人员非常气愤，项目经理竟然要增加一种新的会议，而且他似乎也感觉很不舒服，因为每天都要在会上被问及自己的工作情况。

那么这里到底出了什么问题？是开发人员不讲道理，还是项目经理要求太多？为什么这样一项被广泛接受的简单实践会引发一场冲突？

项目经理和开发人员双方各执一词，言之凿凿。项目经理面临的最大挑战之一是项目规划，这需要投入大量的精力。如果在构建软件时遇到问题，那么团队就有可能偏离当初的计划。项目经理必须努力了解每个人的工作情况，这样才能调整计划，帮助大家解决问题。



图 1-1：一个项目经理想要在团队中开展每日站立会议，却惊讶地发现，并不是所有人都立即表示赞同

而从开发人员的角度看，每天的各种会议会频频打断手头的工作，导致任务很难完成。他知道如何构建自己的代码，不需要另一个人在他耳边唠叨什么计划和变化。他只想一个人静静地写代码，他最不愿意做的事就是再多开一个会议。

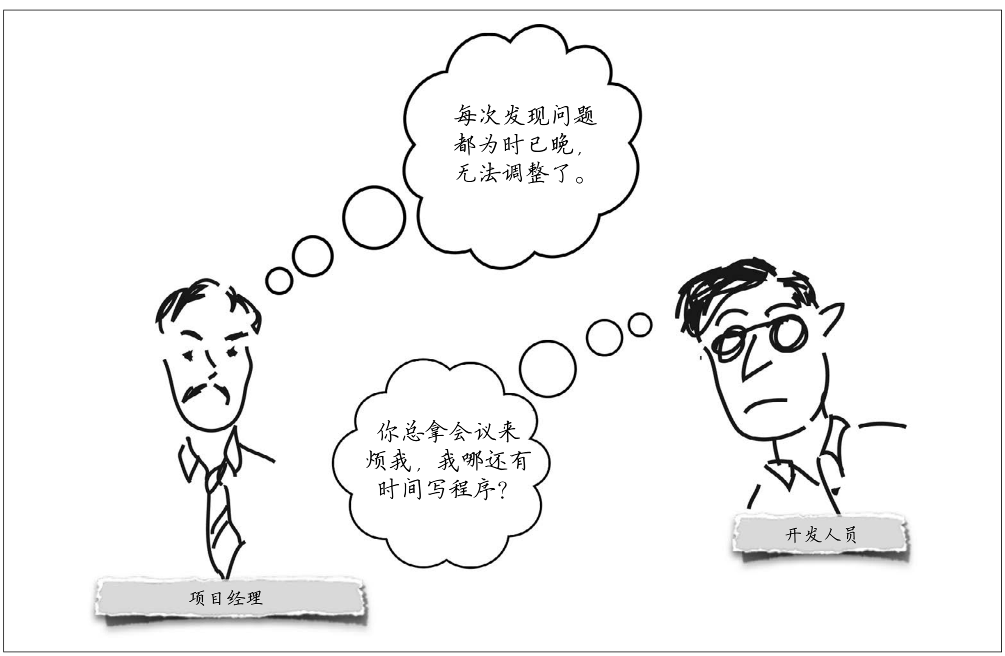


图 1-2：两个人似乎都有充足的理由解释自己为何如此看待每日站立会议。这将对项目产生什么影响呢

设想一下，假设这位项目经理能说服所有人（包括上面那位有抵触情绪的开发人员）参加每日站立会议，那么这个会议会是什么样子？项目经理主要关注实际执行情况与最初计划的差距，因此他会关心每个人的工作情况。而开发人员则希望会议尽快结束，所以根本不注意听其他人汇报，只等自己开口汇报，而轮到自己的时候，则尽可能地长话短说，快点结束。

我们应该清楚，这就是许多每日站立会议的现状。这样的每日站立会议并不理想，但仍然有成效。项目经理能看出自己的计划存在什么问题，而且从长远看，开发人员也能从中受益，因为对他们确实有所影响的问题可以由此尽早解决。从总体上看，这利大于弊，值得付诸实践。

那么要是开发人员和项目经理的思维模式不同会怎么样？如果团队中每一位成员对每日站立会议的态度都截然不同会怎么样？

例如，如果项目经理希望团队中每一位成员都参与到项目计划中，会怎么样？项目经理会真心倾听每一位团队成员，不仅仅是为了检查团队成员是否偏离了自己的计划，还要尝试理解群策群力下的计划可能需要怎样改变。项目经理不会简单地将计划丢给团队并下达命令，然后考核团队执行计划的程度，而是与团队成员一起找出完成项目的最佳方式。每日站立会议就是这样一种合作方式，这有助于确保所有成员时刻保持高效。项目的状况每天都会变化，而团队可以利用每日站立会议作出最有效的决策。由于团队每天都会碰头，所以会议上讨论出来的计划变更可以立即应用于实际工作，这就节省了在错误的道路上渐行渐远的时间。

如果开发人员觉得开会不仅报告了状态，还可以理解当前项目的进展，并且可以每天与大家一起探讨如何更好地工作，会怎么样？果真如此，每日站立会议对他来说就会非常重要。优秀的开发人员不仅对自己的代码有想法，而且常常对整个项目的发展方向怀有见解。通过每日站立会议，他可以确信项目正在以合理高效的方式运行。他还可以确认，从长远看自己的编码工作可以得到更好的回报，因为项目中的其他部分也在正常运转。他还会知道，如果他在会议中对计划提出问题，那么其他所有人都会倾听，项目也会因此更健康地进行下去。



图 1-3：如果团队中每一位成员都觉得自己在项目规划和运行中享有同等地位，那么每日站立会议就会变得更加有价值，而且更加有效

换句话说，即使团队中所有成员都认为“每日站立会议就是一种不得不一起做的进展汇报”，这仍然值得实施，尽管它只比传统的进度报告稍微有效一点儿。要是所有团队成员都明白，每日站立会议的目的是保证每一位成员都在正轨上，大家都在为同一个目标努力，而且每个人都可以对项目的运转方式发表自己的看法，那么每日站立会议就会高效得多，而且所有人都会更加满意。开发人员应该明白，每日站立会议从长远看对他自己和整个团队都是有帮助的。而项目经理应该明白，当团队成员齐心协力完成计划的时候，项目可以得到更好的结果。有了正确的态度，每日站立会议就可以帮助大家更高效地团结协作，更直接地沟通，更轻松地把工作做好。

上面这个例子展示了团队的思维方式和态度对于接纳敏捷实践的巨大影响。本书的一个重要目标是帮助你理解团队的思维方式会对项目和敏捷实践产生怎样的影响。通过探究 Scrum、极限编程、精益和看板方法，你可以从原理和实践两方面学习敏捷，还可以学会如何综合利用这些敏捷方法，构建更好的软件。

1.2 本书的读者对象

你和你的团队有没有遇到过下面描述的这些情况？

你尝试了一种敏捷实践，但是并没有成效。也许你们确实实施了每日站立会议，现在团队每天都会碰一次头，但你还是被各种问题所困扰，理不清头绪，而且无法按时完成任务。也许你已经开始编写用户故事，并且与团队和利益干系人一起审核这些用户故事。但是你手下的开发人员仍然在手忙脚乱地添加特性，应付不停变化的需求。也许你的团队尝试采用 Scrum 或极限编程之类的方法进行大规模敏捷化，但似乎总是感觉华而不实，好像所有人都做了必要的事情，但是项目得到的改善却微乎其微。

又或许你们还没有开始尝试敏捷，但是你已意识到你的团队正面临严峻的挑战，而你不知道从哪着手。你希望敏捷方法能帮你应付那些总是在改变想法的严苛用户。用户每次改变需求，你的团队都需要做更多的工作，最终得到的代码打满了补丁，像意大利面一样纠缠不清。于是软件越来越容易出错，也越来越难维护。你的项目可能一团混乱，最终能够发布全靠加班和逞强，而你希望能通过敏捷给团队指一条出路。

你可能是一位担心手下团队无法交付重要项目的高管。你也许听说过敏捷，但是并不知道敏捷到底是什么意思。你能不能直接命令手下的团队采用敏捷？或者说你是否需要跟着团队一起改变思维方式？

如果你面临上述情况中的任意一种，而且想改进团队工作的方式，那么本书可以帮助你。

本书会解释敏捷方法的理念，包括敏捷方法的产生、能解决的问题，以及包含的价值观、原理和思想。本书会讲解敏捷的过程，进而讲解其着眼点，让你能够找出相关原理，以解决团队、公司和项目中遇到的问题。本书会教你如何利用这些信息选择正确的方法和实践。

本书还适合一类人，那就是*敏捷教练*（agile coach）。公司和团队越来越依赖敏捷教练来指导他们采用敏捷方法和实践，以及为团队中每一位成员树立正确的思维方式。敏捷教练可以通过本书学习一些方法，借此更好地与团队沟通敏捷思想，应对日常敏捷训练的挑战。

1.3　本书的目标

我们希望你可以通过本书达成以下目标。

- 理解高效敏捷团队背后的思想动力，以及整合这些思想所需要的价值观和原则。
- 了解最流行的敏捷流派（Scrum、极限编程、精益和看板），理解这些各不相同的方法如何实现敏捷。
- 学会可以立即用到项目中的具体敏捷实践，同时熟悉高效实践所需的价值体系和原则。
- 更好地理解自己的团队和公司，进而选择与你自己的思维方式匹配（或尽可能匹配）的敏捷方法。让你和你的团队开始学习一种新的思维方式，从而向更高效的敏捷团队迈进。

这些不同的敏捷方法和实践对于构建更好的软件到底有什么帮助？为什么这些方法可以让团队更好地处理变化？为什么这些东西是敏捷的？这些具体的做法，比如在规划的时候使用索引卡或开会的时候站着，真的有用吗？对于刚刚踏上敏捷之路的人来说，这些问题很难回答，而且往往让人感到困惑。读完本书之后，你就会有自己的答案了。

查找讨论敏捷软件开发的博客和文章，你可能首先会看到这样的观点："敏捷开发好，瀑布式开发不好。"为什么敏捷开发好，而瀑布式开发不好？为什么这两种方式互相冲突？有没有可能让一个采用瀑布式开发方法的团队敏捷起来？读完本书之后，这些问题也会有答案。

1.4　努力建立敏捷思维

本书名为《学习敏捷》，因为我们真切地希望你学会敏捷。过去二十多年，我们一直在与为真实用户开发软件的团队一起工作。过去十多年，我们也一直在撰写与软件开发相关的图书（包括 O'Reilly Head First 系列中非常成功的两本书，分别是关于项目管理和学习编写代码的）。借助这些经验，我们发现了很多介绍复杂概念和技术概念的方法，可以让枯燥的内容变得平实易懂。

我们尽量让这些材料有趣且吸引人，但这里仍然需要你的配合。下面来介绍一下本书通篇使用的教学技巧，它们可以帮助你牢牢记住书中的知识。

- 故事

  回想一下你读过的上一本技术书。你能不能回想起那本书涵盖的所有主要内容，以及这些内容的顺序？你可能做不到吧。那么再想想你看的上一部电影。你能不能回想起那部电影中的重要情节，以及这些情节的顺序？你可以做到。这是因为大脑天生擅长记忆引起情感共鸣的事情。本书充分利用了人脑的这个特点，以叙事的方式展开，故事中有人，有对话，有冲突，由此表现人们遇到敏捷时的真实反应。这些人会碰到问题。

  我们需要你做的是：想象自己和书中人物遇到了同样的情况，这样可以与敏捷思想建立起情感连接，从而更容易记住并理解这些概念。对于书中的小故事，请保持开放的心态。如果你不喜欢看小说，那就更要如此。在每一个叙事场景中你都可以学到真正的知识，而这些知识就是本书的核心。

- 图示

  不同的人会采用不同的学习方法。有些人是视觉型学习者，这些人看到真实画面的时候更容易记忆。我们希望为你提供尽可能丰富的学习工具，因此本书中穿插了大量的图示。在某些情况下，我们会大量使用视觉隐喻，例如通过不同的几何形状来表示不同的特性，或者通过齿轮的形状表示复杂的软件。

  我们需要你做的是：如果不是视觉型学习者，你可能觉得有些图示是多余的，而某些图示根本没有意义。但这对你来说是一个很好的学习机会。如果真有这种感觉的话，你应该花一分钟时间思考一下视觉型学习者能从图示中学到什么。这有助于更深入地理解概念。

- 重复

  大部分技术书都会采用这样的模式：首先介绍一个概念，然后详细阐述这个概念，接下来讲解下一个概念。这种方法可以有效地在一本书中尽量塞入更多的信息，但并不契合人脑的工作方式。有时候，人们需要反复接触才能真正理解一个概念。因此，本书会在一章或几章中反复提到某个概念。这种重复是有意为之，目的就是帮助你深入理解概念。

  我们需要你做的是：当你第二次或第三次看到某个概念的时候，可能会想："前面不是提过这个概念了吗？"没错，确实提到过，能注意到这一点说明你很棒！其他的读者可能没有注意到，而你也很难发现每一次重复。这些重复的目的就是为了帮助你学习。

- 由简入繁

  有时候，为了更轻松地理解一个复杂的主题，你可以先了解一些浅显的知识，然后再深入进去。本书便时常采用这种方式：介绍新概念时，首先讲解一个简化的版本（当然，此处并无原理性漏洞），之后再进行详尽的介绍。这种方式有两个层面的作用。如果已经深入理解了这个概念，那么你可以认出这种简化，并且在情感上有反应，这可以让你保持参与。另一方面，如果你还不了解这个概念，那么简化的版本可以帮助你入门，为接下来深入的描述打好基础。

  我们需要你做的是：如果觉得有的内容过于简单，先别急着跳过，也不用担心我们避重就轻或是忘记了重要的内容。你很可能会在继续阅读的过程中发现正在寻找的内容。你可以把复杂概念的简单介绍看作某种"Hello, World!"程序。这些内容可以让不熟悉这些概念的读者受到一定的鼓励。他们会有步入正轨的感觉，并且为后面更深入的学习打下基础。

- 对话式的口语文风

  为了使内容更加吸引人，本书通篇采用轻松随意的文字风格。我们在写作中使用了幽默，时不时抖一些文化包袱。有时候我们会用"我们"和"你"这样的代词，表示我们在直接与你对话。这种做法背后是有科学依据的，认知研究[1]表明，如果感觉自己在进行对话，那么你的大脑就能记住更多东西。

注 1：如果你想知道对话式文风如何有助于学习，可以看一看 Ruth C. Clark 和 Richard E. Mayer 的著作 *E-Learning and the Science of Instruction*。

我们需要你做的是：尽管大部分人觉得随意一点并没有什么问题，但是有些人的确讨厌这种风格。例如，有的读者看到缩写就会气愤，还有一些人看到随意的文风会觉得内容不够权威。我们理解这些顾虑。不管你信不信，你很快就会接受本书的风格。

- 要点回顾
  每一章都会随时总结刚刚讲解的要点。这样可以帮你掌握所有的知识，不会遗漏重要的概念。你的大脑也可以借机稍事休息。

  我们需要你做的是：不要跳过“要点回顾”部分。花点时间看看这些要点。看看能不能回忆起其中的每一个点。如果想不起来的话，往前翻几页增强一下记忆。

- 常见问题
  我们花费了大量时间与为真实用户构建真实软件的团队一起工作，也花了很多时间做关于敏捷的演讲，还与很多人沟通过。在沟通的过程中，有一些问题是大家反复提到的。

  我们需要你做的是：阅读每一章末尾的“常见问题”部分。思考一下，这里列出的问题你是不是遇到过？如果是，你觉得这个答案怎么样？你也许不喜欢我们给出的答案，但是请思考一下，并找出其中的真理。如果那不是你遇到的问题，也请思考一下为什么会有人问这样的问题，这种思考可以帮助你从不同的角度理解书中的内容（在第 2 章，你会明白为什么在团队中这种思考非常重要）。

- 现在就可以做的事
  最有效的学习方法就是付诸实践！每一章的结尾都给出了现在就可以动手去做的事情，包括你可以独立完成的，以及需要与团队合作的。

  我们需要你做的是：显然，你最好真的动手去做！不过现实情况是，并不是每一个团队或公司都对此持开明态度。因此，本书要教给你的最重要的一件事情就是：不要推行与团队思维方式相抵触的实践，这样做不会有好的结果。在尝试动手之前，你要首先考虑一下团队会有什么反应。这与实际动手一样重要。

- 更多学习资源
  牛顿曾经说过：“我能看得更远，是因为我站在巨人的肩膀上。”在这样一个时代写作本书是幸运的，因为现在已经有了众多关于敏捷软件开发的开创性图书。在每一章的末尾，我们会给出一些与该章内容相关的参考资料。

  我们需要你做的是：不断学习！本书全面概述了 Scrum、极限编程、精益和看板方法，但是我们不可能涵盖这些方法的所有细节。本书介绍的大部分思想都不是我们想出来的；幸运的是，你可以向想出这些方法的人学习。

- 教练技巧
  敏捷教练是帮助团队学习敏捷的人。本书为学习敏捷的人而编写，同时也可以作为一份指南，帮助有经验的敏捷教练向团队介绍这些概念。如果你是一位敏捷教练，那么可以在每一章末尾找到教练技巧。教练技巧可以让你理解我们使用的概念和方法，并且帮助你在自己的团队中使用这些方法。

我们需要你做的是：即使不是敏捷教练，你也应该读一读教练技巧，因为教别人是一种有效的学习方法。如果是第一次学习这些概念，你可以想象一下自己要怎样利用这些教练技巧帮助团队更深入地学习敏捷。

1.5 本书结构

本书讲解了高效软件团队的价值观和原则、包含这些价值观的学派以及具体实践，以此帮助你理解敏捷。

下面两章讲的是接受敏捷思维方式所需要的价值观和原则。借助这两章所介绍的方法，你可以判断团队和公司是否准备好了接受敏捷，以及哪些敏捷方法适合你的团队而哪些方法可能很难实施。

- 第 2 章阐述了敏捷的核心价值观。我们会展示软件开发项目团队和困难作斗争的例子，帮助你识别问题的主要来源，也就是“割裂的视角”。我们会讲解敏捷价值观，并且通过一个隐喻帮你理解这些价值观如何统一整个团队的视角。
- 第 3 章介绍了敏捷团队在进行项目决策时要遵循的原则。我们会通过一个实际的软件开发项目的例子，展示每一项原则背后的目的和思想。

接下来的 6 章讲解最流行的敏捷学派：Scrum、极限编程、精益和看板。学完书中讲解的基本应用，你就可以在团队中进行实践。

- 第 4 章讲解了流行的敏捷方法 Scrum，并且说明了自组织团队如何工作。我们给出了在自己的项目中引入 Scrum 的方法，还给出了帮助团队学会自组织的工具。
- 第 5 章展示了 Scrum 团队规划项目时采用的具体实践，以及如何利用这些实践在团队中整合力量，交付有价值的软件。我们会揭示 Scrum 价值观与团队及公司文化的契合程度，这决定了 Scrum 的接纳程度。我们还会谈到不契合的情形，以及应对方法。
- 第 6 章会教给你极限编程的主要实践及其价值观和原则。你会了解它们如何有助于每一位团队成员建立正确的思维模式，以便写出更好的代码：不要憎恨变化，团队中的每一位成员都要学会拥抱变化。
- 第 7 章讲解了极限编程的最后三个主要实践，以及如何利用这些实践避开严重的代码和设计问题。你会了解所有极限编程实践如何构成一个生态系统，帮助构建更好、更易维护、更灵活以及更可变的软件。
- 第 8 章讲解了精益方法，以及建立精益思维方式所需要的价值观。我们会讲解如何利用精益思考工具帮助团队发现并消除浪费现象，以及如何对构建软件的系统建立全局观。
- 第 9 章介绍看板方法及其原理、看板方法和精益方法的关系，以及看板方法的实践。你会明白强调流和排队论的看板方法如何帮助团队将精益思考用于实践。你还将学会如何利用看板方法帮助团队建立一种持续改进的文化。

思维方式、方法和学派并不是敏捷世界的全部。很多公司越发依赖敏捷教练帮助团队接受敏捷方法。因此本书的最后一章献给敏捷教练。

- 第 10 章讲的是敏捷训练：团队怎样学习敏捷方法；敏捷教练怎样帮助团队改变思维方式，以便更轻松地采用敏捷方法；敏捷教练怎样帮助你和你的团队变得越来越敏捷。

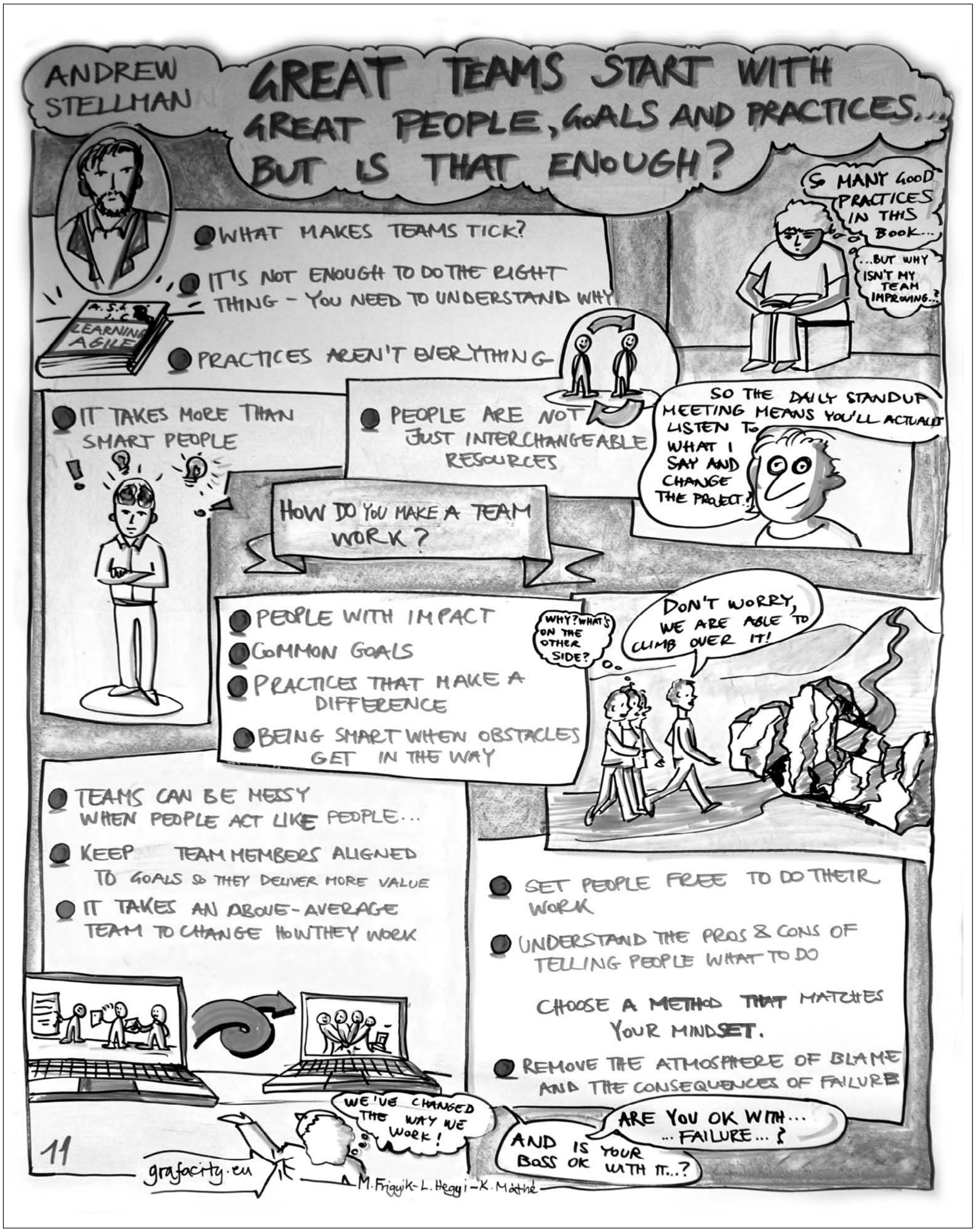


图 1-4：Andrew Stellman 在 STRETCH 2013 会议（匈牙利布达佩斯）上演讲的现场图记。这个演讲的内容基础就是本章

图记：Kata Máthed 和 Márti Frigyik
http://www.remarker.eu

第 2 章

理解敏捷价值观

我们并非因为拥有美德和优点而行事正确，而是行事正确让我们拥有美德和优点。不断实践的行为会决定我们成为什么样的人。优秀不是一种表现，而是一种习惯。

——亚里士多德，《尼各马可伦理学》

作为一次变革运动，敏捷不同于以往的软件开发方法，它从思想、价值观和原则入手，形成了一种思维方式。拥有这样的思维方式，你会成为更加敏捷的实践者，也会成为更有个人价值的团队成员。

敏捷运动在软件开发的世界中是一次颠覆性的变革。采用了敏捷的团队会发现自己构建优秀软件的能力不断增强，有时候甚至会有巨大的飞跃。成功采用了敏捷方法的团队能构建更优秀、更高质量的软件产品，而且开发速度也比之前要快。

业界正处于迈向敏捷的转折点。敏捷方法不再是一种不入流的方法，而是成为了正规的学派。在敏捷方法出现的头几年，采用了敏捷方法的人费了很大力气才让自己的公司和团队相信它可行且值得实践。现在，没有谁会质疑敏捷方法在软件开发上的高效。事实上，2008 年，一项重要的调查[1]表明，超过一半的受访团队正在遵循敏捷方法进行实践，或是遵循了一部分敏捷原则。敏捷正是从那个时候开始流行的。此外，越来越多的敏捷团队已经不满足于通过敏捷解决自己的问题，而是开始考虑怎样在全公司范围内推广敏捷开发。

不过敏捷的推广并非一帆风顺。很多公司在运作软件开发项目的时候习惯于采用*瀑布式流程*（waterfall process）。在瀑布式流程中，团队首先定义好软件的需求，对项目进行完整的规划，然后开始软件设计，接下来再进行代码编写和产品测试。多年来，大量的软件（既包括非常好的软件，也包括垃圾软件）都是通过这种方式构建的。但是几十年来，不同公

注 1：Forrester 2008 全球敏捷公司在线调查。

司的不同团队都遇到了相同的问题。因此有人开始怀疑，项目失败的根源也许就是瀑布式流程本身。

敏捷的故事始于一小群创新者，他们聚在一起试图找到解决这些问题的新方法。他们一开始就对以下这四则价值观达成了一致，认为这些是成功团队和成功项目共有的特质。他们把这四则价值观称为“敏捷软件开发宣言”（Manifesto for Agile Software Development）。

- 个体和互动高于流程和工具
- 可工作的软件高于详尽的文档
- 客户协作高于合同谈判
- 响应变化高于遵循计划

在本章中，我们会学习这些价值观，包括其来源、意义以及如何将它们应用到自己的项目中。我们会对一个厌倦了瀑布式开发的团队进行追踪，介绍在团队成员还没有真正理解如何应用这些价值观的时候是怎样初步尝试实现敏捷开发的。当你阅读他们的故事时，请思考为什么更好地理解这些价值观可以帮助他们避免问题。



故事：有一个开发流式音频点唱机项目的团队

- Dan——开发主管和架构师
- Bruce——团队主管
- Joanna——新招进来的项目经理
- Tom——产品所有者

2.1 团队主管、架构师和项目经理走进了一间酒吧……

Dan 在一家开发投币游戏和信息亭软件的公司担任开发主管和架构师。他参与过很多项目，从制作弹球街机到开发 ATM 机。在过去几年中，他与团队主管 Bruce 合作。他们负责开发公司最大的一款产品：一款名为 Slot-o-matic Weekend Warrior 的 Vegas 老虎机。

Joanna 是几个月前招进来的项目经理，负责领导一款新型流式音频点唱机的软件开发。公司希望在酒吧和餐馆推销这款产品。她在这个项目中非常重要，因为她是公司挖来的，她的老东家开发了一款风靡市场的点唱机。她已经与 Dan 和 Bruce 相处得非常好了，而且十分期待与他们一起开始的新项目。

与 Joanna 相比，Dan 和 Bruce 对这个新项目没有多少精神头。有一天他们一起出去喝酒，Bruce 和 Dan 聊到，团队私底下把老虎机项目叫 Slog-o-matic Weekend Killer，可见新产品并不被团队看好。

在这家公司，项目失败居然是规律而不是意外——发现这一点，Joanna 感到不舒服。公司的经理声称最近的三个项目是成功的，但是这些项目能发布靠的全是 Dan 和 Bruce 疯狂加

班。更糟糕的是，他们故意在代码里偷工减料，导致现在的支持工作成为了噩梦。例如，他们为了某个特性而仓促地做了一个原型补丁，然后就推到产品中了。事后证明这里有严重的性能问题，因为有部分代码根本不能向上扩展。

在他们聊天的过程中，Joanna 找出了其中的模式，明白了导致问题的原因：公司采用的是特别低效的*瀑布式流程*。采用瀑布式流程的团队尝试尽可能早地对要开发的软件写出一份详尽的描述。一旦所有的用户、经理和主管都同意了软件要实现的功能（需求），就会给开发团队递交一份包含这些需求的文档（需求说明书），然后开发团队就动手构建说明书中描述的软件。完成之后，测试团队介入，验证软件与文档是否匹配。很多敏捷实践者把这种方式称为*前期的详细需求分析*（Big Requirements Up Front，BRUF）。

不过拥有多年项目经验的 Joanna 也知道理论和实践的差距，尽管有一些团队可以实施非常高效的瀑布式流程，但是有很多团队却为这个流程而感到痛苦。她开始感觉她所在的这个团队就是众多感到痛苦的团队之一。

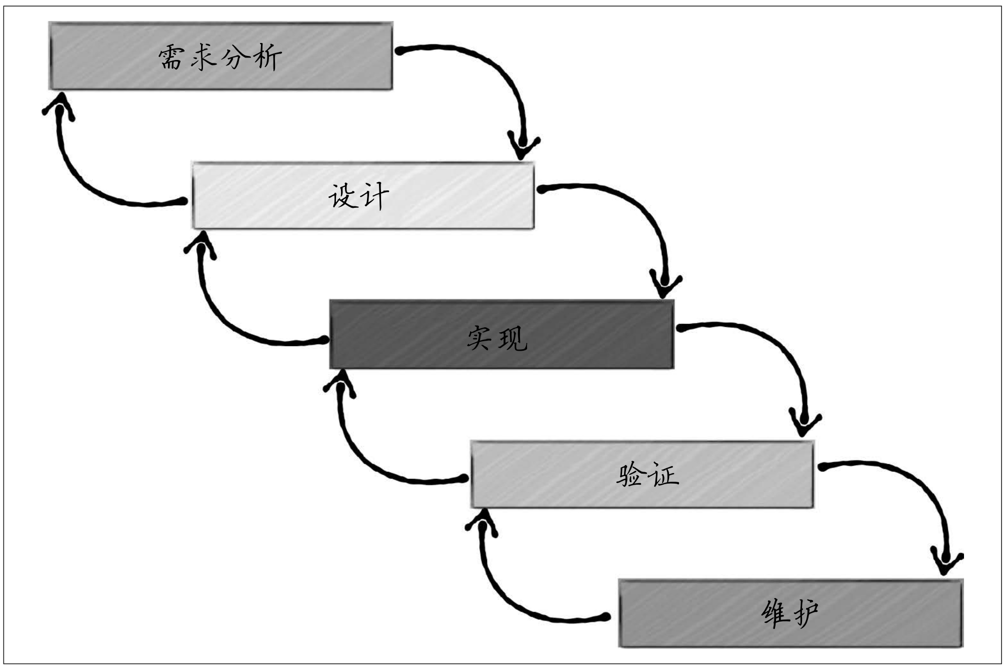


图 2-1：瀑布式模型

在他们聊天的过程中，Bruce 和 Dan 又聊到了一些事情，进一步验证了 Joanna 的观点。Joanna 怀疑公司里有无数说明文档藏在文件夹里蒙尘已久。从某种程度上看，所有人都期望一组用户、经理和主管给出一个完美的需求说明文档。实际上，需求说明书经常变化。到达开发团队手中的那一刻，需求说明书可能已经不准确了。当开发团队完成软件构建的时候，需求说明书可能已经面目全非了。Bruce、Dan 和公司里的很多人都知道完美的需求说明书并不存在，但是他们在运行项目的时候仍然假设需求说明书就是完美的。

时间越来越晚了，Bruce 也越来越放松（还有点小醉意）。他提出了另一个困扰他的问题：在公司里，他工作过的很多团队在构建软件的时候实际上存在很多困难。即使用户十分罕见地提出了准确的需求，即便整个团队破天荒地通过需求说明书完美地理解了这些需求，他们也常常会使用很糟糕的工具，而且在软件设计和架构设计上遇到了很多困难。结果，团队总是开发出有很多 bug 的软件，而且往往混乱而不可维护。

有很多项目的失败都是因为这两个原因，如果把需要长时间工作（每周 90 小时）的项目和充满 bug 的代码称为失败的话，那么以上两个问题就更能解释失败的原因了。Joanna 解释说这些失败的最主要原因就是公司采用的瀑布式流程无法应对变化。在理想情况下，瀑布式流程没有任何问题，因为在项目初期大家就精确地知道项目结束的时候需要造出个什么东西。这样的话，他们可以在一份优雅的需求说明书中把一切写下来，然后交给开发团队。但是项目开发其实从来不会这么美好。

Dan 和 Bruce 现在真的醉了，开始向 Joanna 进行马拉松式抱怨。Dan 说，他工作的项目几乎每一个都遭遇过客户中途反悔，总要开发和最初计划不同的功能。这样，所有人都要回到瀑布模型的最初状态。根据团队采用的严格瀑布流程，他们要从头写一份新的需求说明书，然后进行完全不同的设计，接下来再构建一个全新的计划。实际上没几个人这么做，而且人们极少抛弃已经编写的所有代码，因为时间不允许。相反，他们通常会在现有代码的基础上商定出一个修改方案（hacking）。这样的修改容易产生 bug。在为某一个需求而设计的代码上匆忙修改，使其满足其他需求，这种做法往往会得到混乱交织的代码，如果团队处于压力之下，则更是如此。根据新的需求重写代码会浪费宝贵的项目时间，因此他们最后都会采用糟糕的替代做法（workaround），编写出不可靠的代码。

这一夜结束的时候，Dan、Bruce 和 Joanna 开始意识到导致他们项目存在问题的原因——过分严格的文档、糟糕的沟通以及代码中的 bug。这些导致项目无法跟上正常的变化。

最后，酒吧的服务生帮他们三位叫了出租车。在离开之前，Dan 表示他抒发了胸中淤积已久的怨气。Joanna 很高兴她对于即将加入的项目有了更好的想法，不过并不乐观。她不知道自己是否能够找到方法来解决这些问题。

2.2 没有银弹

如今我们都知道软件构建没有最佳方法。现在没多少人会去争论有没有包治百病的灵丹妙药，但是业界在 20 世纪很长一段时间都坚信一劳永逸。很多从业人员认为能找到一种高度规范的方法，解决所有人在所有场景遇到的项目问题。很多人认为开发人员只要遵循一定的步骤就可以开发软件了，就好像看着菜谱做菜，或者在生产线上组装产品一样。

（具有讽刺意味的是，软件工程领域被引用最多的论文就是 Fred Brooks 在 1986 年发表的“No Silver Bullet”。在这篇文章中，他对这个不可能的目标下了定论。但是这并没有阻止人们去寻找象征着灵丹妙药的“银弹”！）

有不少人对这些类别的问题提出了银弹解决方案。这些解决方案通常都逃不出两种形式：一类属于*方法*（methodology），号称可以为团队提供一套防误操作的软件开发方案；另一类属于技术，据说可以让程序员防止或消除 bug。这背后的思想就是：如果一家公司决定

采用某种方法或技术，那么所有的团队都要遵循公司的正统规范，这样就能源源不断地开发出优秀的软件了。

Dan 和 Bruce 以亲身经历证实这是不可能的，因为他们多年来在公司的项目管理中提出了很多方法和技术，但是都没有带来真正的长期改进。公司每一次寻找软件开发流程银弹的尝试都让所有人失望。Bruce 和 Dan 尤其感到烦恼，因为他们被要求遵循不断变化的流程，而他们并不希望流程发生变化。

Joanna 在职业生涯中也常常碰到类似情况。在上一份工作中，她总是收到一组组严格规定的需求，而且被要求制订计划，将新的需求体现在软件中。接下来她还把自己的计划转交给开发团队，要求严格执行。像这种“做计划，然后执行计划”的团队常常开发出丧失时效性的软件，即使是在部署当天也可能让用户觉得不实用。

让 Joanna 犹豫的是，她工作过的一些团队确实可以通过严重依赖前期文档的瀑布式流程发布优秀的软件。她在一家开发医疗设备软件的公司中用瀑布式实践管理了她最棒的几个项目。

瀑布式流程确实也有用。事实上，如果你明确知道想要做什么，那么在初期将需求写下来是构建软件非常有效的方式。Joanna 的医疗设备软件项目是少有的需求在一开始就很明确的例子，这种项目在实施期间需要做的改动非常少。

可是稳定的需求并不是瀑布式流程成功的唯一因素，这类项目总会遇到很多问题。可以采用瀑布式流程开发优秀软件的团队通常都有以下特点。

- 沟通顺畅。在要求使用瀑布式开发的公司里，成功的团队都会在整个项目期间不断地与用户、经理和主管沟通。
- 实践得力。代码审查和自动化测试阶段的工作尤其有效，可以在流程中尽早发现 bug。人们通常把这个过程称为缺陷预防（defect prevention）。这要求团队主动去思考 bug 最开始是怎样引入代码的。
- 多数文档很少打开。团队成员都明白，写下计划这件事情本身（以及在制订计划的过程中提问）比盲目严格执行计划要重要得多。

这里还要说到一点。Dan 的职业生涯是在 20 世纪 90 年代软件开发工具和技术变革之后开始的，因此，他工作过的团队都会以面向对象的开发技术设计更好的软件。Dan 在工作中还会使用版本控制系统、自动化测试工具、包含自动化执行代码重构和类似设计功能的集成开发环境（Integrated Development Environment，IDE）以及其他一些革新的工具辅助代码的编写。Bruce 参与项目的时间比 Dan 要早，他目睹了团队中的开发人员不断采用新工具自动化一些重复性的日常任务。Bruce 和 Dan 通过实际项目经验认识到，最成功的项目会充分运用优良实践、工具和思想，将时间节省下来，与用户和同事沟通。他们应当思考如何解决问题，而不是花时间去与代码作对。

事实证明，在高效运转的瀑布式项目中，团队成员遵循的价值观、原则和实践往往与敏捷项目异曲同工。那些使用了一些敏捷技术和方法但是没有真正遵循敏捷价值观和原则的项目，通常都会遇到困扰瀑布式项目的问题。

遗憾的是，Bruce、Dan 和 Joanna 将要通过痛苦的方式意识到这一点。

要点回顾

- **瀑布式流程**要求团队在项目开始的时候完整地写下软件的描述，然后完全按照写下的文档构建软件。
- 瀑布式流程很难应对变化，因为这种流程关注的是文档而不是协作。
- 没有什么银弹流程或实践可以让项目完美运作。
- 有些团队能够成功使用瀑布式流程，那是因为采纳了高效的软件开发实践和原则，尤其是加强了沟通。

2.3 敏捷可以拯救乱局吗

即便是第一次听到“瀑布”这个词，你大概也能明白瀑布式流程是个什么样子。[2] Joanna、Bruce 和 Dan 同样心里有数。在计划点唱机项目之前，他们在一起讨论了瀑布式流程在以往团队中是怎样出问题的。

在上一个项目中，Bruce 和 Dan 与公司的客户经理 Tom 共事。Tom 花了大量时间参观游戏厅、赌场、酒吧，并帮助客户安装和使用其产品。在项目初期的三周里，他们三个坐在一起确定新投币机的需求说明。Tom 只有一半时间在办公室，他不在的时候，Bruce 和 Dan 便设计软件并规划架构。等他们三位都对需求达成一致，便请公司的 CEO 和高级经理开会审核，进行必要的修改，获得开工的批准。

彼时，Tom 继续四处奔走，剩下的工作就交给 Bruce 和 Dan 了。他们俩把项目分解为任务，派发给团队中的其他人，然后大家开始编写软件。整个团队快要完成软件构建时，Tom 把商业用户、项目经理和公司主管汇集到一间大会议室，然后演示快要完成的 Slot-o-matic Weekend Warrior 软件。

项目演示并没有像大家期待的那样顺利进行。

在演示的过程中，CEO 问了一个问题：“看起来很棒，但是这个软件不是应该有一个视频扑克的模式吗？”然后气氛就开始变得尴尬了。很明显，CEO 一直认为他们正在开发的软件应该既可以部署在投币机的硬件上也可以部署在视频扑克机的硬件上。高级经理、董事会以及两个最大客户已经多次讨论过这个问题。很可惜的是，没有人想到要告诉开发团队。

最糟糕的是，如果 Dan 在项目早期就知道了这件事，那么改变开发方向也不会太困难，但是到了现在这个节点，他们只能忍痛抛弃大量已经写好的代码，补上从视频扑克项目改过来的代码。他们花了数周时间处理代码整合带来的诡异 bug。Dan 多次在深夜向 Bruce 抱怨说这完全是可预见的，把某个项目的代码仓促地用到另一个项目几乎总会发生这种事

注 2：如果你是项目经理或者准备过 PMP（Project Management Professional，项目管理专业人员）考试，那么你肯定学习过瀑布式流程的全部内容。公平地说，瀑布式流程有其重要性，因为 PMP 认证并不仅限于软件开发领域，而是横跨很多行业，涉及多种方法。如果你要建造一栋摩天大楼或是一座桥，那么在初期制订完整的蓝图往往非常重要，纵然在建造过程中会有变化。

情。现在他要处理一大堆混乱的意大利面条式代码。基本代码的维护也会非常困难，整个团队都感到很懊恼，因为这显然是不该发生的事情。

面临问题的还不只是 Dan 和 Bruce。这个项目的项目经理感到非常不快，因而离开了公司。他曾经信任整个团队的预估和状态，忽然出现的视频扑克要求毁了这一切。整个团队根本不知道他们还要去处理这种意外的硬件变更问题，项目经理的日子也不好过。尽管发生了这么大的变故，但是项目的截止时间仍然雷打不动。到项目结束的时候，最初的计划已经完全过时，基本上就是一份失效的安排，但是项目经理无论如何也要对这份计划负责。遭到公司高级经理斥责之后，他只能辩解说手下的团队在项目初期并没有做好合理的规划。很快，项目经理就离开公司找了另一份工作，接下来 Joanna 就受聘来到了公司。

Tom 可能是所有人中最窝火的，因为客户遇到问题的时候，他总是首先被推到风口浪尖。这个产品最大的客户是 Little Rock 公司，这是拉斯维加斯的一家赌场。这家赌场希望所有的老虎机都能自定义主题，这样他们就可以在阿肯色州的各个城市使用这些产品。客户希望游戏可以变换，而不需要移动游戏主机，并且对新特性提出了要求。他们的工程师后来一直都在与 Bruce 团队留下的 bug 作斗争，这也意味着 Tom 和 Dan 需要花数周时间在电话里与工程师商讨补丁和替代解决方案。尽管 Little Rock 并没有取消合同，但是他们把下一个大单留给了竞争者。而 CEO 和经理却把损失归咎于 Slot-o-matic Weekend Warrior 项目。

最后，所有人都知道项目出问题了。而每个人都有很好的理由把责任推给其他人。但是似乎没有人有什么好点子可以解决这类经常重复出现的问题。他们最终交付的软件也是一团糟。

2.3.1 引入敏捷，带来变化

Tom 再次回到市里，Bruce、Dan、Joanna 还和他一起吃了顿午饭。发完了旧项目的牢骚，Joanna 提议采用敏捷方法。

像很多团队刚开始接触敏捷一样，他们一开始也会讨论“敏捷”这个词到底是什么意思。对于 Bruce 来说，敏捷指的就是敏捷开发的世界：书籍、实践、训练课程、博客以及实践敏捷的人。对于 Joanna 来说，敏捷意味着“项目能够应对变化”，这就是实现敏捷的具体目标。Dan 认为敏捷就是不要任何文档，直接开始编写代码。而 Tom 完全不知道他们在说什么，不过，令他高兴的是，他们在讨论的时候给他展示了很多具体的示例，看来采用敏捷就可以避免上一个项目的悲剧。

团队中开始“敏捷化”的成员通常都会自学敏捷的技术、实践和思想，这个团队也是如此。Dan 加入了敏捷联盟（Agile Alliance，网址为 http://www.agilealliance.org/），开始接触其他的敏捷实践者。Bruce 和 Joanna 开始阅读敏捷开发和项目管理的相关博客和书籍。他们俩都接触了一些很棒的思想，并且以身作则地使用自己学到的方法，期待能解决项目中的问题。他们都学到了不同的技术，并立即开始组合使用这些技术。

Dan 已经为之前的项目写了自动化的单元测试，但是点唱机项目中还有很多开发人员从来都没有写过单元测试。他开始与其他开发人员一起写单元测试，进行*测试驱动开发*（test driven development）。他编写了一个*自动化的构建脚本*（automated build script），设置了一

台构建服务器（build server），这台服务器每小时都会签出一次代码，构建软件，并运行测试。自动测试确实生效了！他们立即发现了代码中一处需要改进的地方。每天都会有一名开发人员发现一个 bug，如果没有自动化测试的话，这些 bug 永远都不会被发现。很明显，他们省下了数周在现场调试追踪诡异问题的时间。不仅产生的 bug 越来越少，他们还感觉构建的代码改动起来更加方便了。

（如果你还不熟悉包括测试驱动开发在内的这些实践，那也没有关系。本书会教给你所有这些概念，并且在第一次出现新的实践的时候用黑体标出，以方便你快速识别。）

Joanna 参加了 Scrum 训练，现在整个团队都开始叫她 Scrum 主管，尽管她在训练课程中了解到 Scrum 主管与项目经理之间有巨大的区别，而且她也不能百分之百地确定自己在项目中的角色真的算是 Scrum 主管。她帮助团队将项目分解为多个迭代（iteration），在任务板（task board）上跟踪迭代的进度，用上了项目速度图（velocity chart）和燃尽图（burndown chart），保证每位成员都能了解最新的情况。燃尽图是一种线图，记录每天项目中未完成的工作，“燃尽”到零意味着工作完成。这是团队成员第一次真正对项目经理做的事情感兴趣，而且这种做法确实改善了项目的进度。

Tom 也想加入敏捷化改造的过程中。Dan、Bruce 和 Joanna 称 Tom 为产品所有者（product owner），Tom 开始编写用户故事（user story），这样可以帮助团队更好地理解他们要开发什么样的软件。根据用户故事，他和团队一起制订构建发布计划（release plan）。现在他感觉自己直接掌控着整个团队要构建的产品。

最棒的是，Bruce 开始每天与 Joanna、Dan 以及所有程序员召开每日站立会议。Tom 也开始加入他们。一开始大家感觉有点尴尬，但是随着项目的推进，大家渐渐能够自在而诚实地交流进度，并知晓项目进度的真实评估情况。Bruce 说服大家在每一轮迭代结束时开回顾会议（retrospective），他欣然发现团队成员在努力实现发言时提出的改进。

2.3.2 “聊胜于无”的结果

所有的努力都开始有效果了。整个团队都在改善，项目的处境也越来越好——一定程度上确实如此。

在向敏捷靠拢的过程中，团队中每一个人的工作都变得更为顺利。Dan 和开发人员开始养成更好的代码编写习惯。Joanna 随时都可以掌握项目的进展。Tom 与团队的沟通也比以前多得多，这让他可以更好地控制开发团队构建的软件，更好地交付用户所需要的软件。Bruce 可以专注于专业技能的改进并进行沟通。

但是，他们有没有变成一个真正的敏捷团队呢？

他们引入了很多优秀的实践。其中有很多都是之前实践的改进，所有这些都提升了团队成员的工作效率。这绝对是一种进步。

然而，尽管整个团队的幸福感提升了，而且点唱机项目也比之前的项目进展更顺利，他们还是对自己所接触的敏捷新流程有所保留。例如，Dan 认为尽管团队现在编写的代码质量肯定比以前好，但是他感觉为了赶进度，他在技术上有所牺牲。

Joanna 对于项目的运转方式有了一定的控制权，她对此感到满意。但是把项目分解为小迭

代这种做法让她感觉有点盲目。现在没有一个自顶向下的大计划可以用作路线图，她发现自己越来越依赖于通过每日站立会议获得项目的进展。每日站立会议很有用，但是在每日站立会议上，大家只是陈述自己的状态，然后 Joanna 忠实地将这些内容记录下来并汇报给利益干系人。她越来越感觉到自己只是一个协调者或组织者，而不是控制整个项目的项目经理。她只关心当前的状态，因此她感觉很难看清项目会遇到的障碍，也无法帮项目铺平道路。团队更擅长应对变化，但是 Joanna 的处境比较尴尬，因为她只能关注如何应对，而无法做全局规划。

Tom 现在已经成了项目负责人，他控制团队构建需求的能力增强了，他对此感到非常欣喜。但是他也很纠结，因为他觉得大家似乎希望他能整天和团队在一起。他必须参加这些每日站立会议，还要不停地回复开发人员关于软件产品细节的邮件和问题。有时候，他也不知道如何回答这些问题，还有的时候他希望他们可以自己解决这些问题。他自己本来就有别的工作要做，他也不喜欢被打扰，他感觉好像其他人把构建伟大软件的责任都推到了他身上，而他自己也不可能解答所有问题。毕竟，他的本职工作是客户经理，那些点唱机又不会自己推销自己。如果整天都在回答程序员的问题，他哪还有时间去了解客户和用户的最新需求？

Bruce 很满意提高之后的团队交付频率。但是回头审视进展，他总感觉有一些事情不尽如人意。他也说不出个所以然。目前，项目毫无疑问得到了改善——他之前的项目都处在失败的边缘。对于 Bruce 来说，引入敏捷有利于项目的进展，没有人被迫逞强，也没有了那么多加班的周末和长夜。但是他也觉得引入敏捷也带来了别的问题。

团队成员，特别是团队主管也有着同 Bruce 类似的感受：采用敏捷的初次尝试让他们有点失望。他们读过的博客和书籍以及参加的培训都提到了“惊人的结果”和“生产力超强的团队”。而目前，点唱机项目相比之前的项目的确有所改善，但是团队绝对没有感受到“生产力超强”，也没有人对产生的结果感到惊喜。

大家普遍感觉项目的状态从不正常转变为了正常，这很好。我们称这样的结果为*聊胜于无的结果*（better-than-not-doing-it result）。难道敏捷方法就这么点能耐吗？

2.4　视角割裂

只要开发软件，团队就会遇到各种各样的问题。事实上，在 20 世纪 60 年代，人们就曾公开讨论过这样一种思想：软件开发从根本上就是有问题的。在 1968 年的 NATO 软件工程大会上，人们提出了*软件危机*（software crisis）这个词。而*软件工程*也是在这个大会上提出的[3]。“软件危机”指的是 20 世纪 70 年代和 80 年代各大公司普遍存在的一种软件开发状态，导致项目失败的各种严重的问题（现在是大家熟知的问题）在当时非常普遍。随着时间的推移，业界开始理解软件危机的主要原因。Lockheed 公司的一名工程师 Winston Royce 在 1970 年发表的一篇论文有着里程碑意义。这篇论文描述了一种非常流行但是非常低效的开发模型。这就是 20 世纪 80 年代广为人知的瀑布式模型。又花了 10~20 年的时

注 3：详见 Peter Naur 和 Brian Randell 编辑的 *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*。

间，大家才真正开始避免盲目地采用这种模型。与 Bruce、Dan、Joanna 一样，很多团队发觉敏捷实践可以解决典型瀑布式流程中的问题，但实施过程不如想象中那么顺利。

开发人员每天都会使用各种软件工具来编写代码。熟练使用多种工具的开发人员比不怎么会使用工具的同行更受欢迎。因此，刚一接触敏捷，很多开发人员立即把这看作一组工具、技术和实践。几乎所有实践了几个月敏捷方法的开发人员都会在自己的简历上更新一笔，添加上他使用过的敏捷实践。这样的初步印象对于敏捷而言还是不错的，因为好的印象可以让本来不关心敏捷的开发人员提起兴趣。

但是看到工具、技术和实践只是步入敏捷的第一步，而且这种做法有一个严重的副作用。首先从 Dan 的角度考虑：作为开发人员和架构师，他的关注点与开发直接相关，也就是能帮他避免 bug、提升代码质量的技术，能快速构建且易于运行的工具，以及帮他改进代码设计、审读以及构建的实践。项目经理 Joanna 更关心的是开发软件所需的工作量以及最终软件的质量，因此她关注有助于理解和沟通进度、预算以及工作量的工具。像 Tom 这样的商业用户在意的则是软件的商业价值。吸引他的主要是可以帮助团队理解用户真实需求的实践，这样团队才能开发出有价值的软件。团队主管 Bruce 期望确保团队中的每位成员都朝着同一个目标奋进，能够进行良好的沟通，而且可以从过去的经验中学到东西，所以他要的是能在这些方面提供帮助的工具。

以编写用户故事这项敏捷实践为例，一个用户故事通常只有几句话，写在一张索引卡上，描述一项非常具体的用户需求。用户故事有时会采用严格的格式，有时却很灵活。例如，点唱机项目中的一个用户故事是这样的："我是一名酒吧常客，我希望能播放当日发行的最新热门歌曲。"

团队中每一位成员都会从不同的角度看待用户故事。

- Joanna 是一名要努力成为 Scrum 主管的项目经理，在她眼里，一个用户故事就是一件要完成的任务，应该整齐地打包好并随时可以构建。她把每一个用户故事都写在索引卡上，并把这些索引卡贴在白板上，通过这种方式让大家不偏离正轨。
- Dan 是开发主管和架构师，在他眼里，用户故事用简单易懂的方式描述了一项要实现的功能。他可以把用户故事分解为小的任务，并且为每一个任务创建一张索引卡。当他开始着手完成某项任务时，就会把自己的姓名写在对应的卡片上。当他完成某项任务时，则将对应的卡片转移到白板上专门摆放已完成任务的区域。
- Tom 是产品所有者，在他眼里，每个用户故事都可以给公司带来价值，因为他可以通过用户故事清晰地看出团队正在开发的软件与用户使用之间的关联。用户故事可以帮助他与客户更好地沟通，让他更加了解客户需要什么样的点唱机软件。他要确保每一个用户故事都准确描述用户的一项需求。
- Bruce 是团队主管，在他眼里，每一个用户故事都是团队要完成的一项目标。他帮助团队成员计划下一步要完成的事情，并且通过进度让整个团队保持动力。

这样在团队中引入用户故事可以改进团队开发软件的方式，因为以上四种角色中的每一种都可以通过用户故事改进自己的工作方法。

但这也有可能产生副作用。Dan 过去的项目都有一份详细的规格说明书，没有多少操作灵

活性。而现在，他可以自由参与构建软件的相关决策。这是好事，但也会在项目中引入一些问题。当 Dan 在为“最新热门曲目”用户故事编写代码的时候，他想到的是编写一项功能，让酒吧顾客播放任何最新上传到服务器上的热门歌曲。Tom 不得不解释说，在点唱机上播放较新的歌曲意味着酒吧老板必须支付更高的版税。Tom 向团队解释了这个用户故事的细节，也就是让顾客播放最新热门歌曲的频率达到令人满意的程度，但又不会导致超支。Dan 对此非常沮丧，因为这意味着他不得不重写这项功能的大部分代码。而 Tom 也很生气，因为这意味着这份软件第一次发布的时候不会带上这项功能。

如果 Dan 一开始就能理解用户故事对于 Tom 验证用户真实需求的价值，那么他在动手编码之前就会找 Tom 讨论一下软件要开发成什么样子。另一方面，如果 Tom 肯花一点点时间，就会发现 Dan 参考用户故事这样有限的信息开发软件。明白了这一点，Tom 会在这一轮迭代开始之前找 Dan 讨论上述需求。但是他们并没有进行这样的对话，这个项目遇到了之前瀑布式项目也会遇到的问题：开发人员作出了错误的假设，然后开始编程，最后不得不对代码做一些本可以避免的修改，而这些修改使得软件更有可能出错。

如果每个人只考虑自己的工作，只关心用户故事如何帮助自己，而不进一步看一看整个团队可以怎样使用用户故事（或其他的敏捷工具、技术和实践），那么最后就有可能出现这样的问题。我们把这种问题称为*视角割裂*（fractured perspective），因为每个人对敏捷实践都有不同的看法。

现在我们把流式音频点唱机项目放一边，本书不会再讨论这个项目了。他们能否解决问题并交付软件？在你阅读本章剩余部分的时候，请尝试找出能帮助他们解决问题的方法。

2.4.1 视角割裂带来的问题

如果项目团队中的每一位成员都只从自己的角度看待一项实践，老问题就会重复出现。在软件危机的年代，开发人员会在还没有花时间理解用户需求的情况下着手开发。他们总是在软件开发中途发现突如其来的新需求，因此不得不删掉和替换一大堆代码。很多敏捷实践的目标都是帮助团队成员在项目初期充分理解客户的需求，由此避免低效工作。如果团队成员不主动沟通——例如，开发人员在还没有真正讨论好用户需求的时候就开始闷头写代码，然后把写好的代码丢给“墙外的”产品所有者——这种工作方式会频频产生需要事后修补的问题。

与此同时，产品所有者很高兴敏捷方法为团队指明了用户需求的正确方向。产品所有者本来感觉对项目缺乏控制，只能无助地看着开发团队开发去年的软件而不是新软件，因为程序员最后一次与用户沟通的时间就是去年。对于有这种感觉的产品所有者来说，敏捷方法会带来极大的宽慰。但是如果发现团队开发的软件并没有完全与他编写的用户故事匹配，产品所有者仍然会感觉很受挫。开发团队感觉产品所有者似乎希望大家可以读懂他的心思。产品所有者则感觉所有人都盼着他成天泡在团队里，回答一切问题。

项目中的其他角色也会遇到同样的视角割裂现象。项目经理和团队主管很高兴看到开发人员可以自主地添砖加瓦。他们看到了改进，但是工作的方式并没有发生根本性的变化，因为团队成员的工作可能相互抵消，导致整体工作方式没能真正改变。如果项目经理把钉在白板上的用户故事看成 Microsoft Project 甘特图文件的一种直接替代品，或者仍然一心想

“指挥和控制”项目，那么为了严格服从原始计划，这名项目经理会经常要求大家加班应对项目变化。团队主管可能会有防御式的反应，例如为了避免团队加班工作而拒绝更紧张的时间安排，要求更宽松的期限，或是减少工作内容。项目经理和团队主管似乎都没有做错什么。如果在一开始都能看到对方与自己视角不同，那么他们也许可以避免冲突，同时也能收获好的结果。

换句话说，如果整个团队不进行沟通，那么就算团队中的每一位成员都采用了某一种工具（例如用户故事），他们也仍然保持老态度，因而会出现摩擦和团队问题。新的工具更为先进，因此项目运转更流畅。但是团队成员感觉变化并不大，因为很多老问题依然会出现。这时大家会开始怀疑敏捷不过如此。

有证据表明，很多团队都经历过这种问题，即单独地采用了一些“聊胜于无”的工具。VersionOne 是一家开发敏捷软件工具的公司，这家公司还通过很多其他途径为敏捷社区作出过贡献。他们做的最重要的事情之一就是每年进行一次“敏捷开发状态”（State of Agile Development）调查。根据 2013 年的结果[4]，我们很容易看出，很多团队确实通过引入敏捷做出了一些改进。

- 2013 年的 VersionOne 敏捷开发状态调查中，88% 的参与者表示他们的组织采用过敏捷开发实践。
- 92% 的参与者表示，该调查评估的每一个领域同去年比都有改进。其中，改进效果最好的几个领域包括：优先级变化的管理（92%）、生产力的提升（87%）、项目透明度的提升（86%）、团队士气的提升（86%）以及软件质量的提升（82%）。

尽管开发团队可以看到敏捷项目进展更快，而且团队成员应对变化的能力更强了，但是敏捷项目仍然会失败，原因通常都关乎瀑布式开发与敏捷方法之间的文化和理念差异。调查参与者将“缺乏使用敏捷方法的经验”“公司文化与敏捷价值观有冲突”和“要求遵循瀑布式实践的外部压力”三点列为敏捷项目失败的主要原因。

当新的敏捷团队遇到问题的时候，最常见的原因就是他们还没有真正抛弃老的瀑布式思维。例如在前述的点唱机团队中，仅仅是增加具体的实践并不能让他们化解导致冲突的问题，开发过程还是出现了本来可以避免的变更。也许点唱机团队中的每一位成员都认为自己已经步入敏捷了，但实际上，他们在很多方面依然是一个瀑布式团队，只是采用了一些很好的敏捷实践而已（Forrester Research 的 Dave West 为此发明了一个词：Water-Scrum-Fall[5]）。换句话说，他们已经达到了瀑布式团队能够实现的最高效率。

2.4.2 为什么视角割裂只能做到“聊胜于无”

人们能交付的东西通常取决于他们所关注的东西。越是关注自己的目标，而不是整个团队的目标，那么他们能为公司交付真正价值的可能性就越低。

对于尝试走向敏捷的团队来说，这是一个悖论。专注于具体实践的团队最终会得到一些改

注 4：在网站 http://stateofagile.versionone.com/ 上可以获得最新的 VersionOne 敏捷状态调查报告。

注 5：*Forrester* 中 Dave West 的文章“Water-Scrum-Fall Is the Reality of Agile for Most Organizations Today”，2011 年 7 月 26 日，网址为 http://www.storycology.com/uploads/1/1/4/9/11495720/water-scrum-fall.pdf。

善，但是仅限于他们非常擅长的领域，原因在于团队成员只关注他们已经知道的东西。任何人要想大大扩展自己已知的领域，都必须很好地掌控未知的事物。要求一个团队在自己还不熟悉的领域获得进展，这似乎是一种离谱的要求。

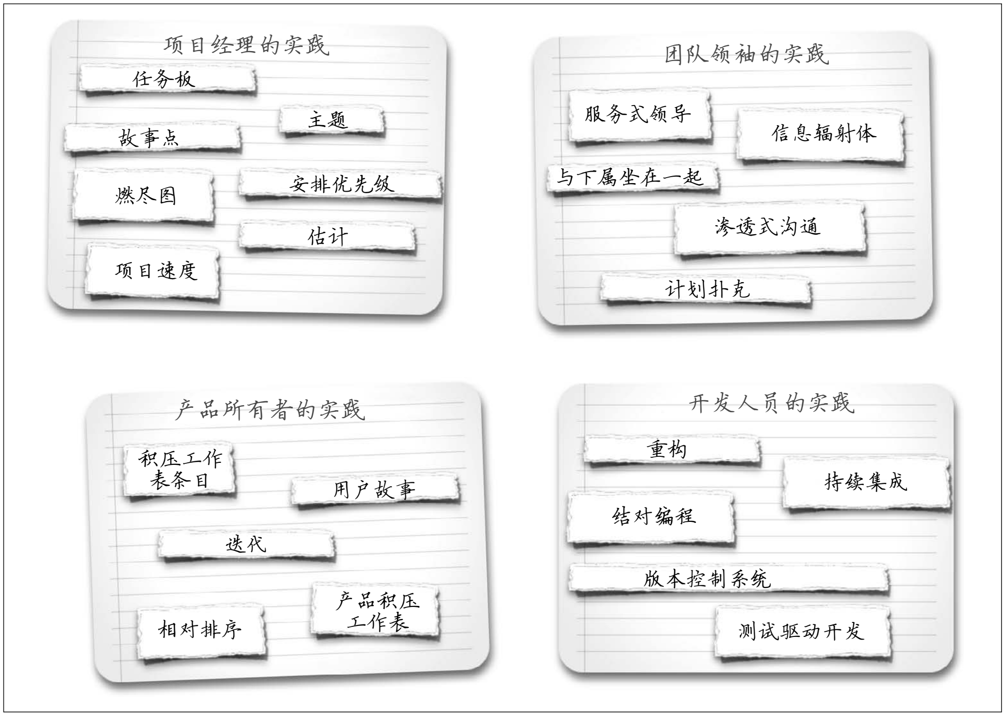


图 2-2：团队成员倾向于在项目中他们已经熟悉的领域采用敏捷实践，因此整个团队只能在这些领域有所改进，这就是视角割裂会导致“聊胜于无”的原因

这也就是为什么一些采用一项敏捷实践的团队通常都只能得到“聊胜于无”的结果。他们只是在擅长的领域采用了更好的实践，因而把这些事情做得更好。但是他们还没有触及项目以往没有得到关注的领域，因为影响这些领域的实践对团队中任何成员都没有吸引力。因此，相关问题都不会得到改善。然而，也许正是这些领域的问题妨碍了团队高效地交付惊人的产品。

团队怎样解决这个问题呢？

要点回顾

- **更好的沟通**可以帮助团队更好地管理变化。
- **团队作为一个整体来制订计划**（planning as a team）好过事先过度详细地制订计划，然后盲目执行。
- 软件项目具有不可预知性，从 20 世纪 60 年代就开始产出不好的结果。这种状况在当时被称作“软件危机”。
- 很多团队“步入敏捷”的方法是**采用伟大的敏捷实践**改善他们已经做得很好的事情。
- 因为团队没有从根本上改变沟通和工作的方法，因此他们采用更好的实践得到的只是**聊胜于无的结果**。
- **用户故事**是一种敏捷实践。在这种实践中，一名团队成员（通常是产品所有者）通过几句话描述用户操作系统的各种具体方式，使用的语言是用户可以理解的语言。
- 目前，采用敏捷最常见的方法是一次采用一种实践，但是这并不是最有效的方法。

2.5 敏捷宣言帮助团队认识实践的目的

敏捷软件开发宣言也称敏捷宣言（http://www.agilemanifesto.org/），由 17 位志同道合的 IT 人于 2001 年在犹他州盐湖城外群山中的 Snowbirt Retreat 旅馆写就。这份宣言意在为他们职业生涯中看到的软件开发问题提供一个解决方案。经过数日的讨论，他们对一组核心思想和原则（以及“敏捷”这个名称）达成了一致。他们把这些内容打包成了一份独立的文档，这份文档开启了软件开发世界思维的转换。

敏捷宣言包含四则简明的价值观。下面是这份宣言的完整内容。

> 我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。
> 由此我们建立了如下价值观。
>
> 个体和互动 高于 流程和工具
> 可工作的软件 高于 详尽的文档
> 客户协作 高于 合同谈判
> 响应变化 高于 遵循计划
>
> 也就是说，虽然右项有其价值，但是我们更重视左项的价值。

为了理解敏捷并高效地使用敏捷，我们首先要解读这些价值观。

2.5.1 个体和互动高于流程和工具

盲目地遵循流程会使人走入误区。好的工具有时候可以帮你更快地犯错。软件的世界充满了各种伟大的实践，并不是所有的优秀实践都适合项目的具体情况。然而，这里有一个普适原则，那就是团队中的成员要心里有数。他们需要理解一起工作的方式，明白每个人的工作会对其他人造成怎样的影响。

对于想要改善自己所在团队工作方式的人来说，这是非常实际的想法。敏捷团队认为个体和互动高于流程和工具，因为仅仅拥有“正确的”流程和“最佳的”实践还不够。如果使用者并不认可他要使用的流程或工具，他就无法将这些东西坚持使用到最后。更糟糕的情况是，人们只是表面遵循这些流程规定的动作，即使这些做法会得到毫不相干的结果。在你打算实施一项流程的时候，即使从逻辑上说这种流程非常合适，而且从理性上看采用这种流程是非常正确的选择，你还是需要把这项流程推销给你的队友们。如果大家不明白你这么做的理由，也不明白你到底在做什么，那么在他们眼里，你只是在随意发号施令而已。

这也就是为什么你必须在任何一种情形下都意识到你是在和一个团队合作。团队中每个人都有自己的动机、想法和喜好。

有很多敏捷实践都支持这条原则，这体现了敏捷思想的诸多特点。因此，在本书中，你可以看到有很多支持个体和互动的实践，例如*每日站立会议*和*回顾会议*。在回顾会议上，大家讨论当前项目或迭代的进展情况，以及可以吸取的教训。*用户故事*也是这样一种实践，故事本身并不重要，重要的是这些故事可以帮助团队一起讨论故事代表的真正意义。

2.5.2 可工作的软件高于详尽的文档

摆满大量软件详细卷宗的架子随处可见。在一个软件项目中，有太多的事情可以文档化。而且在项目升温期，很难判断哪些文档会在未来有用，而哪些文档只能束之高阁。因此，很多团队（特别是很多团队经理）都会决定采用详尽的文档，事无巨细，全部记录下来，也不考虑以后会不会有人读。

相对于详尽的文档，敏捷团队更为重视可工作的软件。但是“可工作的软件”这个词看上去有点含糊不清。这到底是什么意思？对于敏捷实践者来说，可工作的软件是可以给公司组织带来价值的软件。这可以是公司出售的软件，也可以是帮助公司员工更高效工作的软件。如果可以增加价值，那么这个项目能交付的价值或能节省的成本必须比开发项目本身的成本要高。尽管团队不会直接谈钱，但是价值大部分情况下最终都会落实到钱。团队应该着重于构建和交付可以带来价值的可工作的软件。文档只是实现目标的工具。

可工作的软件高于详尽的文档，但文档还是要写的。在一个团队中，有很多种类的文档非常有用。但是必须注意的一点是，撰写文档的人通常就是编写软件的人。好文档能帮助团队理解问题，与用户沟通，以及避免将错误的需求开发进软件。这种文档所消耗的成本与节省的时间和精力相比是划算的。程序员也不介意编写别的文档，例如线框图和时序图，遵循上述原则就好。

从另一方面看，关注可工作的软件能够确保团队没有偏离正轨。如果清晰地表明了可工作软件的方向，那么这种文档就对项目有贡献。实际上，团队通常可以采用一些将文档嵌入

软件本身的创新方法。测试驱动开发就是这样一种敏捷实践。在测试驱动开发中，程序员首先开发自动化的单元测试，然后再开发上述单元测试测试的软件。自动化的测试代码和软件本身的代码并列保存。自动化的测试也可以当作文档使用，因为测试可以帮助程序员记录代码应该完成的功能，以及软件中单个组件预期的行为。

2.5.3 客户协作高于合同谈判

看到"合同谈判"，很多人可能会认为这只与咨询顾问以及合同制的承包商有关。实际上，在一个公司内部，有很多团队也需要接触这类事务。如果程序员、产品测试人员、产品所有者和项目经理都在不同团队中工作，而且不是真正朝着交付可工作的软件这样一个单一的目标而合作努力，那么他们的工作方式就好像是互相遵照合约合作。在很多公司中，不同开发团队之间、测试和开发之间以及开发团队和用户之间都会把服务级别协议（Service-Level Agreement，SLA）放在台面上讨论。

这样做也许可以降低风险，减少与老板之间的矛盾，因为你可以指责其他团队影响了软件的交付。但是如果大家要达到的目标是给公司外的用户交付可工作的软件，这种做法只会适得其反。一名开发人员如果总是想办法保护自己，那么他就不太愿意尝试新的合作方法，也不愿意为需要他们开发的软件的用户采用创新的方法。

敏捷团队落实这项价值观的一种方法是在团队中安置一名产品所有者。这名产品所有者可能不会参与代码的开发，但是他会参加会议，贡献想法。最重要的是，他要把最终的产品当作自己的东西。产品所有者通常通过用户故事与团队中的其他成员合作。

2.5.4 响应变化高于遵循计划

项目管理中有一句老话："怎么计划怎么来。"遗憾的是，如果计划有误，那么构建出来的产品就是错误的产品。因此，开发团队需要不断地发现变化，当用户需求发生变化，或者软件构建方式需要变化的时候，团队要保证正确地响应变化。如果环境变化了，项目就需要新的计划。

制订计划的人抗拒变化是很常见的事情，因为改变计划需要消耗精力。例如，要把工作分割成多份，并估算每一份的工作量，这本身就需要消耗不少精力。一个变化就可能导致项目经理把这些事情全部重做一遍。如果他认为遵循计划高于响应变化，那么他可能会自己陷入困境中。尽管遵循计划有利于项目顺利执行，但是如果真的有变化出现，在代码完成度更高的时候处理变化更为困难。

任务板（task board）是一种良好的实践，可以帮助团队作出响应变化的正确决策。工作中的每一个要素（比如用户故事这样的典型要素）都写在一张索引卡上，并且贴在一块大板上，这块大板通常分为多个列，展示每一个工作要素的不同状态，Joanna 在点唱机项目中就是这么做的。任务板也可以通过计算机程序管理，但是很多团队觉得在一面真实的墙面上摆放这些索引卡效率更高，因为大家可以站在任务板前讨论、指点和移动用户故事。这些沟通方式比单纯的讨论要丰富得多。任务板这样设置，大家都可以重新规划任务的顺序，甚至为此跃跃欲试。如果发生了变化，那么大家可以往任务板上添加索引卡，而不是通过某一个项目经理来澄清一切。这样可以帮助大家跟上进展，让计划不断更新。

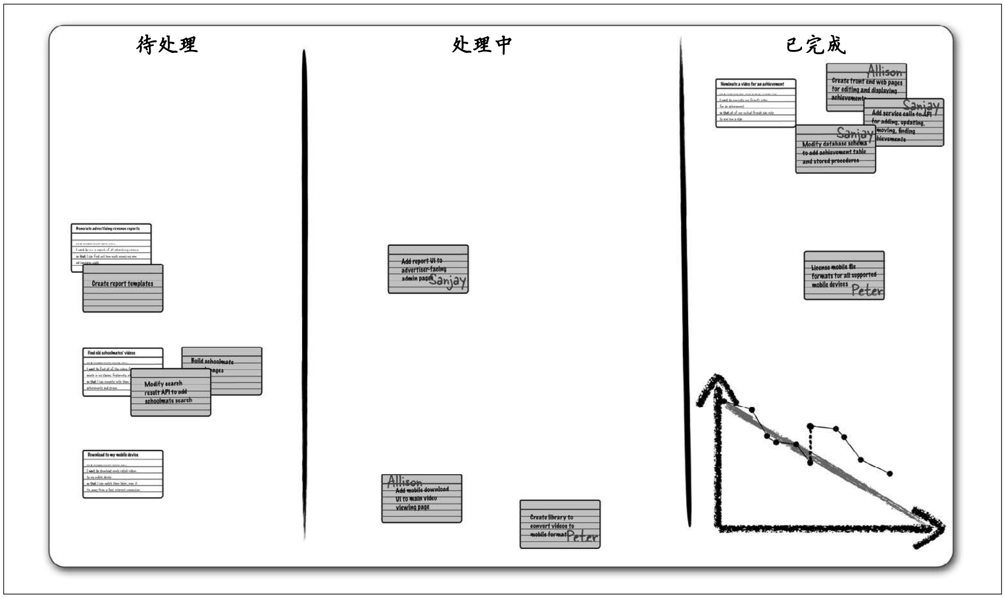


图 2-3：敏捷团队通常会使用任务板来展示任务并跟踪进度。他们会把任务或用户故事写在索引卡上，然后根据项目的进展移动这些卡片。很多团队还会在任务板上画图跟踪进度

2.5.5 原则高于实践

点唱机团队采用了一些优秀的实践，所以有了很不错的收获。这些好的实践对项目是有改进作用的。但是这个团队视角割裂，没有全体全力朝着构建软件目标奋进，所以也受益有限。在实践之上，还有一套敏捷的思维。那些发现了敏捷方法背后的思想的团队都找到了更好的合作和互动方式。

换句话说，在通过互动、协作以及对变化的积极响应来构建对客户有价值的软件时采用敏捷实践，团队会比仅仅在计划、编程和文档编写上采用收获更多。

Jim Highsmith 在他的著作《敏捷项目管理（第 2 版）》中进行了非常好的总结。

> 没有具体的实践，原则是贫瘠的；但是如果缺乏原则，实践则没有生命、没有个性、没有勇气。伟大的产品出自伟大的团队，而伟大团队有原则、有个性、有勇气、有坚持、有胆量。

那么，一个团队怎样才能超越引入实践的简单做法，变得“讲原则”，从而开发出伟大的产品呢？

要点回顾

- **敏捷宣言**包含了高效团队必备的共同价值观和思想。
- “个体和互动高于流程和工具”的意思是说，团队应该首先关注团队中的人以及人之间沟通的方式，工具和实践是次要的。
- “可工作的软件高于详尽的文档”的意思是说，交付满足用户需求的软件比交付描述这个软件的说明文档重要。
- “可工作的软件”指的是可以给公司带来价值的软件。
- “客户协作高于合同谈判”的意思是说,要把所有人看作同一个团队的成员。
- 很多高效的敏捷团队把**产品所有者**当作项目团队中的一员，通过这种方式与产品所有者合作，而不是将其当作客户进行谈判。
- “响应变化高于遵循计划”的意思是说，要意识到计划是会变的，交付软件产品比严格遵循计划更重要。
- **任务板**是一种敏捷规划工具，大家把用户故事粘贴在任务板上，并且根据用户故事在当前项目或迭代中的状态，把这些用户故事分类在不同的列中。

2.6 理解敏捷的“大象”

Lyssa Adkins 在她所著的《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》一书中讲解了比喻对概念理解的重要作用。

> 长期以来，专业的敏捷教练都知道比喻的实用性。事实上，敏捷培训的专业课程会教授这样的核心技能。敏捷教练会提出一些问题，帮助客户建立自己的比喻。这种比喻必须发自内心且能引起共鸣。客户借此整理生活中出现的各种事件。[6]

有一个非常有用的比喻可以帮我们理解视角割裂及其对高效工作的阻碍。下面是盲人摸象的故事。

> 受人邀请，六位盲人触摸大象身体的不同部位，判断大象的长相。摸到大象腿的那位盲人说大象就像一根柱子。摸到大象尾巴的那位盲人说大象就像一根绳子。摸到大象驱干的那位盲人说大象就像一截树干。摸到大象耳朵的那位盲人说大象就像一把扇子。摸到大象肚子的那位盲人说大象就像一堵墙。摸到大象长牙的那位盲人说大象就像一根坚固的管子。
>
> 国王对他们解释道：“你们都没有错。你们每个人都给出不同答案，因为你们触摸的是大象的不同部位。这头大象具有你们每个人提到的特点。”[7]

那些采用了敏捷，却得到“聊胜于无”结果的团队往往在采用敏捷方法之前就可以交付质量不错的软件产品。他们对敏捷方法寄予厚望。问题在于在此之前，他们遭遇了问题。而

注 6：《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》，Lyssa Adkins 著。

注 7：引用自故事 Blind Men and the Elephant 的维基百科页面（网址为 http://en.wikipedia.org/wiki/Blind_men_and_an_elephant，本文加入维基百科的日期为 2014 年 6 月 25 日）。

这些问题并不是十分严重，没有带来毁灭项目的软件危机，只是会给团队带来摩擦和不安。这就是视角割裂：开发人员考虑的是开发人员的问题，项目经理考虑的是项目经理的问题，而他们直接把代码丢给考虑业务问题的业务用户。每个人都忙于自己的项目工作，太专注于自己这一块。谈到团队之间的隔阂和失调，他们常常会说“甩包袱”这样的话。每个人只考虑自己的工作，人和人之间没有太多的沟通。事实上，他们都在为同一个项目单独工作，而并没有真正形成团队。

这里就可以用“盲人摸象”的故事来打比方了。以割裂的视角采用敏捷方法，每个人都只会使用对自己的工作有影响的实践，就好像每一个盲人只摸到大象的一部分。例如，开发人员会关注测试驱动的开发、重构和自动化构建。项目经理喜欢任务板、项目速度跟踪和燃尽图。商业用户通过发布计划和用户故事来更好地了解团队的进度。团队主管通过每日站立会议和回顾会议管理和改进团队。大家都想从项目中得到不同的东西，每个人都只在意对自己有帮助的实践。（再提一下，我们在本书中会学习上面提到的每一种实践，所以如果感觉不熟悉的话也不要担心。）

大家各自采用这些实践肯定能对现状有所改善，因为敏捷实践真的很棒。问题在于开发人员、项目经理、商业用户和团队主管看待项目的角度各不相同。如果每个人只关注对自己有直接吸引力的实践，只看敏捷实践中对自己有用的那部分，并且认为敏捷的意义就在于让其他所有人都围着自己的观点转，那么就会得相反的效果（“看，我一直都是对的！”）。

敏捷这头“大象”由很多优越的实践组成，而整体比零散个体的总和更为强大。如果只看到某个方法，甚至只看到对自己有用的方法，那么你只能认识到敏捷的一小部分。敏捷由各种日常实践组成，但是敏捷本身却远远超越了这些实践。



图 2-4：敏捷的“大象”作为一个整体比具体的各个实践加起来要大

如果成员看到的只是一个个独立的实践，而不去思考背后的原理，那么这个团队就会错过人和人之间重要的互动。不同人的视角会一直处在割裂状态。团队成员互相独立，不能真正地发挥团队力量。尽管依然可以把工作完成，但是他们忽略了团队成员间重要的互动和合作，而这才是敏捷真正发挥力量的地方。

互动已经内建在敏捷方法中了。这里再看一下敏捷宣言中的第一则价值观。

> *个体和互动高于流程和工具*

流程、方法和工具依然非常重要（因此敏捷宣言最后谈到，虽然右项有其价值，但是我们更重视左项）。但是比具体实践更重要的是个体和互动。这些价值观（以及第 3 章要学习的 12 条原则）展示了如何把实践整合在一起使用，也指导了团队的具体工作。

让实践快速上手

理解了敏捷宣言中的价值观（及其背后的原则），你还需要真正改变团队开发软件的方式。幸运的是，在敏捷开发中还有一个重要的方面专门解决这个问题。*敏捷方法*的作用就是帮助团队采用敏捷并改进自己的项目。

敏捷方法非常有价值，因为它可以帮你在具体情境中了解敏捷实践。对于那些不熟悉所有敏捷实践的团队来说，这一点尤为重要。每一种方法都经过了多年的开发和改进，从事相关工作的正是专注于整个敏捷开发方法的专家。采用完整的敏捷方法意味着遵循一种经过检验的可靠路径，从头到尾完成软件项目，避免导致割裂视角的反复试验。

敏捷方法是一组实践的集合，除此之外，这里还包含了相关的思想及建议，以及大量来自敏捷实践者的知识和经验。一种敏捷方法会罗列出项目中所有人的不同角色和职责，并且在项目的不同阶段对每一位成员提供具体的实践建议。

VersionOne 在 2013 年的敏捷开发状态调查报告中列出了一系列最流行的敏捷方法，其中位居榜首的是 Scrum，后面是 Scrum 和极限编程的一种混合版本。调查参与者还谈到了精益方法和看板方法，尽管这些方法并不属于敏捷方法（参见第 8 章和第 9 章），但是仍然可以反映敏捷的核心思想。

Alistair Cockburn 在《敏捷软件开发（原书第 2 版）》中这样描述 Scrum。

> 虽然使用起来另有要点，但我们可以对 Scrum 概括出以下几点。
>
> - 团队及项目出资方在一起创建一个优先级列表，包含这个团队需要完成的所有工作。这个列表称为*产品积压工作表*（product backlog），它既可以是任务的列表，也可以是特性的列表。
> - 每个月，团队取出列表最上面的一部分，这是团队预估的一个月的工作量。团队将这部分工作扩展为一个详细的任务列表，称为*冲刺积压工作表*（sprint backlog）。团队向出资方承诺在月底可以演示或交付上述积压工作表的处理成果。
> - 团队成员每天碰面，花五到十分钟同步进度，交流阻碍。这项活动称为每日站立会议。

- 将一个人指派为 Scrum 主管。这个人要负责亲自或指派他人解决站立会议上提出的问题。[8]

对于很多刚开始采用敏捷的团队来说，这些可以直接对等为具体的实践（在此以楷体字突出表示，第 4 章会详细解释这些实践）。

- 产品所有者创建并维护一个*产品积压工作表*，即软件的需求列表。
- 团队在限定的时间内完成月度*冲刺*（sprint）：在产品积压工作表中找出满足一个月工作量的需求，对这些需求进行开发、测试和演示。当前冲刺对应的需求列为*冲刺积压工作表*。（有一些采用 Scrum 的团队将一个冲刺的时间跨度定为两周或四周。）
- 团队召开*每日站立会议*，每个人都汇报昨天完成的工作以及当天计划完成的工作，讨论在工作中遇到的任何困难。
- Scrum 主管扮演的角色是领导者、教练以及为团队完成项目保驾护航的指导者。

不过采用 Scrum 并不仅仅意味着采用这些优越的实践。这些实践本身都可以以不考虑敏捷价值观和原则的方式使用。比如，对于仅仅以实践协同推进项目的团队来说，每日站立会议运转得很好。但是每日站立会议也可以让项目经理给每个人分配任务，并且知晓每个人当前的状态。在会上，每一位开发人员都会这样向项目经理汇报——这些是阻碍我进展的障碍，你去把它们都搞定吧。如果每个人只顾自己，总认为很多责任是别人的，那么每个人都会把障碍看成别人的问题。这样的会议就变成了讨价还价，而不是一种合作。陷入这种境地的团队可能采用了类似 Scrum 的实践，但是并没有真正地使用 Scrum。

（本书后面会详细讲解 Scrum 的工作方式及其实践。）

第二套方法称为*极限编程*。James Shore 和 Shane Warden 在 *The Art of Agile Development*（http://shop.oreilly.com/product/9780596527679.do）中这样总结极限编程："通过同步推进工作阶段，极限编程团队每周都能发布可以部署的软件。在每一次迭代中，开发团队都会分析、设计、编码、测试并部署一个功能子集。"（很多极限编程团队选择的迭代周期为一周，还有一些团队选择两周或一个月。Scrum 也可以适应多种不同的迭代长度。在本书后面我们会深入学习敏捷方法的使用。）极限编程描述了具体的开发实践，这些实践的目标是增强与用户的合作、计划、开发以及测试。而极限编程则更进一步，通过使用这些实践来帮助团队构建简单、灵活的软件设计，同时方便团队维护和扩展。

Scrum 和极限编程有很多共同之处，例如这两种方法都需要*迭代*。项目要分解为多个迭代，开发团队在每一次迭代结束时产生一个可工作且可部署的软件。在这些迭代的过程中完成整个项目的所有工作。很多极限编程团队使用持续一周的迭代，而很多 Scrum 团队采用的是持续一个月的迭代。对迭代持续的时间设限的称为*时间定量*（timeboxing），这样可以帮助用户了解新特性交付的期望时间。

很多团队，特别是采用了 Scrum 和极限编程的团队，发现采用整套方法比采用实践更为有效。尽管零散地采纳实践可以让团队中每一位成员针对自己的工作作出选择，但是采用完整的方法可以鼓励整个团队齐心协力，帮助团队认识到如何恰当地引入一套方法中所有的实践。为此，团队成员需要改变有关工作的思维方式。敏捷方法都是围绕着敏捷价值观和

注 8：《敏捷软件开发（原书第 2 版）》，Alistair Cockburn 著。

原则构建的，观念的转变也要顺应团队合作、互动、可工作的软件并响应变化。其他敏捷实践者的书籍和知识积累通常可以为这种转变提供帮助，采用这些方法的团体也会树立正面榜样。

精益不是一种方法，而是一组思维方式。它包含一组价值观，以及帮助你接受这套价值观的思考方式。在敏捷领域，精益的重要性不亚于极限编程和 Scrum。理解了这三种方法之间的共性，你就可以充分了解精益对敏捷的意义。看板可以改进团队开发软件的方式。看板方法的构建基于精益价值观，包含了一组帮助团队改进和演化的独特实践。

极限编程方法的实践和关注点与 Scrum 方法在很多方面有不同之处。精益和看板也有不同的实践和关注点，采用了完全不同的方式。既然这些敏捷的方法都采用了完全不同的实践和关注点，这些方法怎么可能都能称为敏捷方法呢？这是因为所有的敏捷方法都基于相同的原则，而且都依赖于团队中的每一位成员齐心协力完成项目中的每一个部分。敏捷宣言中的价值观和原则把所有这些方法和实践绑在了一起。

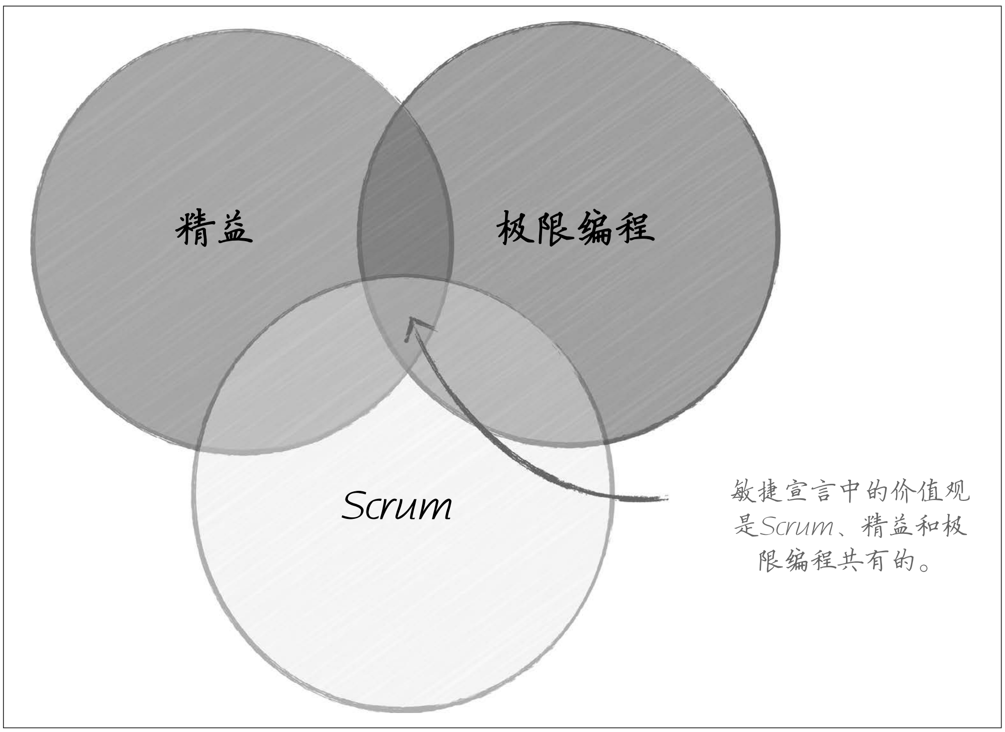


图 2-5：Scrum、极限编程和精益的核心都是敏捷价值观，这些方法之间也会有一些共同的价值观、思想和实践

2.7 着手采用一套新方法

当团队成员一起努力采用一套方法的时候，每一位成员都会探讨实践、思想和视角。这就彻底避免了割裂的视角。把整套方法看作一个整体，团队开始理解单个实践之间如何互

相影响。这是 Bruce、Dan、Joanna 和 Tom 想要达到的境界，但是他们不知道怎样走到这一步。

准备接纳新的实践和思想时，团队成员还无法理解这些新事物与他们已经熟悉的实践有什么关系。这样的理解要等团队收获经验时才能获得。敏捷方法之所以能够起效，是因为它有一套完整的体系，其中包含了一组已知交互良好的实践，而且团队也能利用这些实践提升生产力。采用一组完整的实践是了解实践交互的基础。

不过，采用一套方法比寻找适合团队当前工作方式的实践要难得多。如果可以一次引入整套方法，那么团队就更有可能获得最佳的敏捷效果。从某种程度上说，这是因为除了那些与原先行事风格类似的实践之外，这样还会引入一些大家一开始觉得没有用处的实践和思想。

根据我们之前了解的情况，点唱机团队陷入困境的原因在于 Bruce、Dan、Jonna 和 Tom 各自独立地引入了敏捷实践。要想获得最好的敏捷效果，仅仅引入一些实践是不够的。大家一开始就应该坐在一起认真讨论每一项实践会带来什么样的好处。但这种做法具有挑战性，因为他们不知道怎样开展讨论。和很多团队一样，他们也面临着这样一种困境：如果已经知道这些敏捷实践会给项目带来怎样的效果，并且知道如何真正实施这些实践，那么他们也就不需要这类讨论了。但是他们恰恰还不知道这些。

这个问题也有解决办法。我们需要查看与敏捷宣言所列价值观紧密结合的 12 条原则。我们在第 3 章会学习这些原则。

要点回顾

- 只关注单个实践的团队看不到**加强沟通和响应变化的大目标**。
- **敏捷方法**涵盖了实践、思想、建议和团队。
- 诸如 Scrum、极限编程和精益这样的敏捷方法不仅包含了很多优越的实践，还要求团队关注这些目标背后的思想。
- 敏捷教练通常使用隐喻帮助团队学习。

常见问题

敏捷宣言中谈到不要写详尽的文档，是不是说什么都不要写？

这是一个关于敏捷宣言的常见问题。我们再看一下敏捷宣言中关于详尽文档的描述。

可工作的软件高于详尽的文档

这并不是说敏捷实践者不在乎详尽的文档。而且这句话显然也没有说什么文档都不要写！有很多文档尽管并不“详尽”但是依然很有用。

这句话的意思是说，要让用户认可团队的进展，最好的办法就是把可工作的软件交付出去。不过项目里还有一个地方可以写写文档。我们会通过代码注释给代码添加文档（例如，解释为什么需要做某个决策，或为什么不用另外一种方法或算法实现某个功能）。本书后面还会介绍一种特定的文档形式：*用户故事*。用户故事通常写在索引卡上，可以用来帮助成员、团队、用户以及利益干系人共同准确地了解正在开发的产品。敏捷团队还会使用一些其他类型的文档，其中有一些比另一些更为详尽。

你确定吗？肯定有人跟我说过敏捷的意思就是什么文档都不写，而且什么计划也不做，直接开始编程。这样不是更高效吗？

敏捷团队从来不做计划——这是一种十分常见的误解。事实上，敏捷团队做的计划比很多传统项目团队细致。但是对于新的敏捷开发人员来说，敏捷项目看起来没有多少计划，因为整个团队都在积极参与，没有人抱怨。（说实话，在传统的团队中，收到项目计划会的邀请会让程序员心生不满。）

比如，Scrum 团队通常会花一整天（8 小时）计划 30 天的迭代。然后他们会召开每日站立会议（通常限定在 15 分钟之内），大家一起审阅计划。对于一个 5 人团队来说，这相当于在迭代开始的时候耗费 40 个工时制订计划，另外 40 个工时分布在接下来的 30 天中。这样的计划工作量算下来多于很多传统团队 30 天软件开发中的计划工作量。难怪 Scrum 团队能很好地按期完成任务。而对于团队成员来说，这种计划工作并不会让人感到“无聊”。这是因为他们都参与到这个过程中了，他们关心的是产出，而且他们有理由相信计划能让整个迭代周期进展顺利。（在第 4 章中我们会深入学习 Scrum 项目计划。）

然而，从局外者的角度来看，他们好像没有计划就直接进入项目了。团队只在 30 天迭代周期的第一天做迭代计划，这意味着团队在第二天就可以开始编写程序（如果他们认为这样最合理的话）。这样一来，计划看似没有，但制订计划的零散时间加起来还是很长的。

这是不是意味着只有擅长计划的资深开发人员才适合敏捷开发？

不是的。敏捷开发适合所有技能水平的所有人。计划是一种技能，提升这种技能的唯一方法就是实践。即使是有经验的开发人员有时候也会作出错误的估计（其实他们经常犯错！）。我们真的看到过很多团队的初级开发人员成功采用敏捷开发出远超公司预期的软件。有一点需要注意的是，在高效敏捷团队中，初级开发人员很快就不是初级水平了，这可能是有人认为敏捷只适合资深开发人员的原因之一。

我能不能只让团队中的开发人员采用敏捷，而让其他成员（测试员、业务分析师、用户体验设计师和项目经理等）维持原状？

可以的，但是这样可能会影响效率。当人们谈到只让开发人员采用敏捷方法的时候，他们真实的意思是让这些开发人员采用敏捷方法中的一部分实践。这样可以让开发人员自己的生产力得到提升，因此这么做是值得的（即“聊胜于无”的结果）。但是团队并没有改变思考项目的方式，因此这样采用敏捷，团队得到的改进非常有限。这是团队落入“瀑布式 Scrum”境地的原因之一。团队成员最终会感觉他们采用敏捷的尝试是徒劳的或是不完整的。

如果我没有用 Scrum、极限编程、精益和看板方法，是不是意味着我的团队就不敏捷了？

绝对不是。敏捷方法有很多种，我们这里只关注了其中的几种。本书的目的是通过这些方法讲解敏捷背后的思想。更重要的是，本书还要帮你了解敏捷的真实含义。在本书剩下的部分中，你会学到不同方法的价值观和实践，并且通过这些内容学到敏捷的真实奥义，以及这些迥异的方法如何共同体现敏捷。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 列出你和团队开发软件使用的所有实践。这些实践可以是：编写规格说明书、在版本控制系统中管理代码签入、使用甘特图记录项目计划、开每日站立会议等。
- 请团队中其他人也写一写这样的列表。比较你们的列表。有哪些实践没有被所有人纳入列表？请就此展开讨论。你能不能找出成员间视角的不同？

更多学习资源

下面是与本章相关的其他学习资源。

- 《敏捷软件开发（原书第 2 版）》，Alistair Cockburn 著：深入了解敏捷价值观和原则。
- 《敏捷项目管理（第 2 版）》，Jim Highsmith 著：深入了解原则和实践的关系。
- 《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南 》，Lyssa Adkins 著：深入了解敏捷教练。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 培训一个新团队的时候，单独与团队成员聊天，尝试理解不同角色之间视角的差异。
- 单独询问团队成员对敏捷宣言中价值观的理解：怎样看待这些价值观？哪些价值观最重要？这些价值观是否可以用在日常工作中？

- 团队通常会有一种“聊胜于无”的感觉，但是不知道如何表达。直接提出这个概念，要求团队成员想出一些感到“徒劳”的实践或一些事半功倍的实践。
- 发起关于敏捷宣言中某项价值观或原则的讨论。例如，如果团队谈到了他们与客户之间协商定下的“合约”,那么可以以这份合约作为起点讨论合同谈判与客户协作之间的异同。帮助他们理解在哪里作出选择。

第 3 章
敏捷原则

如果我问人们想要什么，他们肯定会说想要更快的马（而不是汽车）。

——亨利 · 福特[1]

没有什么灵丹妙药可以保证总是开发出完美的软件。敏捷团队也意识到了这一点。一些思想和基本原则可以帮助团队作出正确的选择并避免问题，或是处理那些必然会发生的问题。

我们在敏捷宣言中已经看到了四则价值观。除了这些价值观之外，每一位敏捷实践者还需要在软件项目开发团队中应用 12 条原则。敏捷宣言最初的 17 位签署者在犹他州的滑雪胜地很快就四则价值观达成了一致，但他们在宣言的 12 条附加原则上耗费的时间较长。下面是宣言签署者 Alistair Cockburn 的回忆。[2]

> 17 人的小组很快一致选择了这些价值观。而进一步的陈述却无法在这个会议上敲定。本节描述的这些价值观构成了当前的工作成果。
>
> 我们会越来越了解人们的看法，也会找到更准确的描述，这些陈述应该随之修改。在本书出版之后，如果现在的这个版本没有很快发生变化，我会感到惊讶的。最新版本的描述可以参见敏捷联盟（Agile Alliance）的网站（http://www.agilealliance.org/）。

Alistair 说得没错，目前网站上对原则的描述确实与他书中的描述有所区别。这些描述可能会不断地演化，但是思想和原则一直不变。

在本章中，我们会学习敏捷软件开发的 12 条原则，包括原则的内容、为什么需要这些原则以及这些原则会对项目有什么影响。通过实践中的例子，我们会学习如何将这些原则应用到真实的项目中。为了帮助理解，我们把这些原则分为四个类别：交付、沟通、执行

注 1：亨利 · 福特是否这么说过是有争议的，但是大家觉得这很符合他的风格。

注 2：《敏捷软件开发（原书第 2 版）》，Alistair Cockburn 著。

和改进。这四个类别反映了敏捷原则中恒定的主题，也反映了敏捷开发的一般性原则。不过，这样进行分类只是学习这些原则的一种有效方式，每一个原则都是独立的。

3.1 敏捷软件开发的12条原则

(1) 最优先要做的是尽早、持续地交付有价值的软件，让客户满意。
(2) 欣然面对需求变化，即使是在开发后期。敏捷过程利用变化为客户维持竞争优势。
(3) 频繁地交付可工作的软件，从数周到数月，交付周期越短越好。
(4) 在团队内外，面对面交谈是最有效、也是最高效的沟通方式。
(5) 在整个项目过程中，业务人员和开发人员必须每天都在一起工作。
(6) 以受激励的个体为核心构建项目。为他们提供所需的环境和支持，相信他们可以把工作做好。
(7) 可工作的软件是衡量进度的首要标准。
(8) 敏捷过程倡导可持续开发。赞助商、开发人员和用户要能够共同、长期维持其步调，稳定向前。
(9) 坚持不懈地追求技术卓越和良好的设计，以此增强敏捷的能力。
(10) 简单是尽最大可能减少不必要工作的艺术，是敏捷的根本。
(11) 最好的架构、需求和设计来自自组织的团队。
(12) 团队定期反思如何提升效率，并依此调整自己的行为。[3]

3.2 客户总是对的吗

翻回本章开头，重读一下章首的引文。亨利 · 福特想表达的真实意思是什么？他的意思是要向人们提供人们真正需要的东西，而不是提供他们要求的东西。客户有一个需求，如果你要构建符合这个需求的软件，那么必须理解这个需求，不论他是否能与你沟通。客户无法在项目初始告诉你他实际上需要的是一辆汽车而不是一匹更快的马。那么你应该如何同他打交道呢？

这就是 12 条原则的初衷：让团队构建用户真正需要的软件。这些原则的根基在于这样一个想法：我们要交付有价值的软件。但是“价值”这个词有点难以捉摸，因为每个人都会看到软件中不同的价值，不同的人对软件的需求是不同的。

现在，在你们手中就有一个很好的例子可以解释这一点。如果通过手持式电子书阅读器阅读本书，那么你就在使用能显示电子书的软件。花一分钟时间思考一下这个电子书阅读器软件的不同利益干系人（即对这个软件有需求的人）。

- 作为读者，你希望可以在这款软件上方便地读书。你关注的是软件的功能：前后翻页、高亮段落或记录笔记、搜索文本以及跟踪上次读到的页数。
- 作为作者，我们非常关注我们写的文字能否正确显示，列表项的“小圆点”是否正确缩进以方便阅读，读者是否可以在正文和脚注之间跳转，是否有良好的整体体验。这是享用我们的作品并从中学到东西的基础。

注 3：来源：http://agilemanifesto.org/principles.html（截至 2014 年 6 月）。

- 编辑关心读者能否方便地获得图书，喜欢这本书的读者是否方便给好评并从出版社买其他图书。
- 将本书售卖给你的书商或零售商希望读者便于浏览和购买他们卖的其他图书，并且可以快捷方便地下载电子书。

你还可以想想其他的利益干系人，以及他们各自关心的事情。上述的每一件事情都是软件带给利益干系人的价值。

市场上的第一款电子书阅读器并没有实现上述所有目标。运行在这些阅读器上的电子书软件花了很长的时间才进化成现在这幅模样。几乎可以肯定的是，随着开发团队不断地发觉传递新价值的新方法，电子书软件会越来越好。

就电子书阅读器的软件而言，我们很容易看出其价值所在，因为我们已经能看到了阅读器软件现在的状态。而在项目刚开始的时候，要看出其价值就会困难得多。为此，我们做一个简单的思维实验。考虑这个问题：假设使用瀑布式流程开发，那么这个阅读器会是什么样子的？

“按我现在说的做，而不是按我之前说的做”

假设你在首款手持式电子书阅读器的开发团队中工作。硬件团队交付了一款原型设备，带有一个 USB 接口，可以把书传进去，还有一个小键盘用来交互。现在轮到你和团队开发显示电子书给读者的软件。

遗憾的是，你所在的公司多年以来一直使用“开始前建立完整的需求”这种特别低效的瀑布式开发流程。因此，项目经理做的第一件事情就是找所有人开大会。接下来的几个星期里，你的整个团队都会泡在会议室中，一会儿与公司里的高级经理开会，一会儿与出版电子书的出版社代表开会，一会儿与想要卖书的在线零售商的高级销售人员开会，此外还要与项目经理能想到的所有其他利益干系人开会。

在多日的高强度会议和激烈讨论之后，业务分析师开始把所有信息整合在一起，制作成一份巨大的规格说明书，其中包含了从所有利益干系人那里搜集到的需求。这是一项很庞大的工作，不过现在已经有了一份大家都认为很不错的说明书。这份说明书包含了大量用户特性，足以成就最先进的手持式阅读器软件。这还包含了可以帮助出版商获得市场数据的特性，提供网络书店，方便购书，甚至还有一项创新的特性，能帮助作者在写作的时候预览和编辑自己的书，从而优化出版流程。这份说明书定义的软件的确是一款革命性的软件。然后你和你的团队坐下来预估时间，发现需要 15 个月。尽管这个时间非常长，但是大家都很兴奋，你也相信自己的团队可以交付这样的软件。

让我们看一下一年半之后发生的事情。电子书阅读器团队工作异常勤奋，投入了无数个通宵和周末，还给几段婚姻带来了压力。团队付出了巨大的努力，最终项目还是完成了，而且准确按计划交付了项目，几乎一天也不差。（是的，这看上去是不可能的！但是既然这是一个思维实验，所以就请相信确实发生了这样的事吧。）说明书中的每一项需求都实现了、测试了并且经验证已完成。整个团队非常骄傲，所有看到最终成果的利益干系人都认为这准确地实现了他们想要的功能。

最后，这款产品终于推向市场，结果惨败。没人为这款阅读器买单，利益干系人都不满意。这是怎么回事？

大家明白了一个事实：一年半前需要的软件并不是现在需要的软件。自项目开始以来，业界已经有了新的标准电子书格式。由于没有在规格说明书中体现，所以这种格式并没有得到支持。没有哪家网络零售商愿意出售这款阅读器使用的非标准格式。尽管团队开发了非常棒的网络店铺，但是零售商目前正在使用的店铺要先进得多，因此团队的作品对谁都没有吸引力。另外，你和你的团队花了那么多工夫为作者开发特殊预览功能，而竞品则让作者直接给读者通过电子邮件发送和显示 MS-Word 文档，高下立现，你们输了。

真是一团糟啊！你的团队在项目刚开始的时候制订的软件规格说明书，当时对所有公司内外的客户都很有价值。但是到了现在，一年半之前决定开发的这一款软件价值已经大大降低。有一些变数是在项目初期就可以发现的，还有很多变化在项目开始的时候是不可能预见到的。为了考虑到这些变化，团队应当在项目中的很多时间点快速地改变自己的方向。“预先指定大计划”的瀑布式开发方法限制了团队响应这些变化的灵活性。

那么我们怎样才能让项目更好地满足利益干系人和客户的需求，同时交付可工作软件？

3.3 交付项目

敏捷团队知道他们最重要的工作就是要给客户交付可工作的软件。在本书第 2 章你已经了解了他们达到这个目标的方法：以团队工作，与客户协作，并响应变化。但是团队在日常工作中应当怎么做呢？

如果团队能将交付价值当作首要目标，将变化看作是项目中的好事，并且频繁交付软件，那么这个团队与客户就可以一起工作，在开发的过程中及时调整。团队开发出来的软件不一定与刚开始计划的一样，但这是好事情，因为最终开发出来的软件就是客户最需要的软件。

3.3.1 原则1：最优先要做的是尽早、持续地交付有价值的软件，让客户满意

这条原则包含了三个独立的重要概念：尽早发布软件、持续交付价值，以及让客户满意。为了真正理解这项原则的核心，我们需要知道这三点是怎样结合起来的。

项目团队工作的环境是真实世界，在真实世界中没有什么事情是完美的。即使能够很好地收集并写下需求，团队仍然会漏掉一些需求，因为完美提取任何系统的完整需求都是不可能的事情。并不是说这不值得一试，敏捷方法的基础就是沟通和记录需求的优秀实践。问题在于客户在真正拿到可工作的软件之前，都很难想象软件到底应该如何工作。

这样说来，既然客户只有在看到了可工作的软件之后才可能给你真实有信息量的反馈，那么获得反馈的最佳方式就是*尽早交付*（early delivery）：尽早给客户交付第一个可工作的软件版本。即便是只交付了一个可以工作的特性给客户使用，这也是一种突破。这对整个团队都是有益的，因为客户可以给出有价值的反馈，这样开发团队才能朝着正确的方向推进

项目。这对客户来说也是有益的，因为拿到了软件就可以使用。也就是说，开发团队实际交付了真正的价值。尽管只是一小部分价值，但是比到最后一点价值都不交付要好，特别是在客户因为等了很长时间还没有看到软件而越来越愤怒的情况下。

尽早交付有一个缺点：最初交付给客户的软件完成度非常低。一些用户和利益干系人可能很难容忍这个。有一些用户喜欢尝鲜，但是还有一些用户对过早交付的软件兴趣低得多。很多人对于不完美的软件感到很不舒服。实际上，在很多公司里（特别是花费很多年与软件团队合作的较大型公司），软件开发团队必须认真协商向利益干系人发布软件的条款。如果开发团队与乙方之间没有很紧密的合作关系，而开发团队交付了不完整的软件，那么当用户和利益干系人发现有任何他们希望的功能缺失的时候，会给出非常可怕的处罚。

敏捷的核心价值观对此给出了答案：客户协作高于合同谈判。如果受限于固定的规格说明书，而且需求的变更面临僵化的官僚阻力，那么这样的团队别无选择，不可能让软件随着时间推移而演化。在这种条件下，团队必须启动全新的变更管理流程，这要求与客户重新进行一轮合同谈判。真正与客户协作的团队可以在开发过程中任意进行任何有必要的改变。这就是持续交付（continuous delivery）的意义所在。

敏捷方法通常采用迭代，原因就在于此。敏捷团队选择出能交付最大价值的特性和需求，并据此计划项目的迭代。团队确定哪些特性能交付价值的唯一方法就是与客户协作，并利用前一次迭代收到的反馈。从短期看，团队可以通过尽早交付价值让客户满意，从长期看，交付最终产品的时候可以实现价值最大化。

3.3.2 原则2：欣然面对需求变化，即使是在开发后期。敏捷过程利用变化为客户维持竞争优势

很多成功的敏捷实践者初识这条原则就遇到了很多困难。欣然面对变化说起来简单，但是在项目热火朝天地进行的时候，如果团队要处理需要大量工作的变化，开发人员可能会有情绪，特别是在老板不顾及工作量，要求不改变截止时间的情况下。这个坎可能很难过，特别是在团队因为项目延期而遭到抱怨的情况下。但是跨过了这个坎，收获也很大，因为欣然面对需求变化是敏捷工具箱中最有力的工具之一。

为什么项目中的变化会激起众怒？理解这一点就是理解本条原则的关键。想一下，做项目的时候发现正在开发的东西需求改变了，你会有怎样的感受？在知道需求变更之前，你以为项目进展得很好。你可能已经做了很多决策：如何规划产品结构，要开发什么产品，向客户承诺交付什么。结果现在项目外的某个人突然告诉你这个计划中有些错误，是你出错了。

“你出错了”，接受这样的指责是很难的，如果这样说的人还享受着你的服务，那就更是如此了。大部分软件工程师都是在技术自豪感的驱动下工作的：我们交付的产品我们能负责，而且能满足用户的需求。而项目中发生的变化则对这种自豪感产生了威胁，因为变化是在质疑你采用的方法，质疑你的设想。

经常出现的情况是，别人明确地告诉你怎样做之后却要求你改变方向。假设一个人要求你开发某个产品，你投入工作并完成了一半。如果这个时候，这个人跑过来对你说：“跟你

讲，我经过了一番思考，我们能不能开发一个完全不一样的东西？”这会让人非常沮丧，感觉自己投入的努力没有得到尊重。现在你不得不返回去修改你以为已经完成的工作。人们很难不对此产生抗拒心理。更糟糕的是，你没有读懂客户，但还不得不冲击截止时间。

几乎所有职业开发人员都至少经历过一次这样的情况。在这样的前提下，我们怎样才能让自己欣然接受需求变化呢？

欣然面对需求变化的第一步是尝试从客户的角度看问题。尽管这一点并不总是容易做到，但是这种做法依然很有启发作用。你认为客户之前误导你了吗？当他发现你在他授意下开发的软件不对，导致你浪费了数月时间的时候，你认为他心里是怎么想的？他跑来告诉你需求有变化，这实际上是承认他自己犯了错误，让你白做了很多事情。这对他来说并不容易。难怪客户通常都会过了很长时间之后才跑过来告诉团队说要改变需求！他们知道自己带来的是坏消息，这是很让人难堪的行为。你不能按期完成任务了，他也一样。如果他有需求，而且公司花钱开发软件满足他的需求，而他的需求没有得到满足，那么这个项目就没有产生价值。而这全都是他的错，因为他在项目一开始的时候传递了错误的信息。

换句话说，两方都被要求做不可能的事情。你被要求读懂客户的心，而他被要求预测未来。如果你这么看问题的话，需求变化看上去就好接受多了。从另一方面看，如果你就是抗拒项目的变化，而且就是想严格遵照项目开始时指定的计划执行，也很简单，只要保证团队中的成员都具有心灵感应的能力和读心术就行了。

那么欣然面对需求变化意味着什么呢？它意味着以下几点。

- 不要认为有变化就会有人“倒霉”。我们都承认，而且老板也承认，我们都是人，都会犯错。如果公司允许我们犯错并且能尽快改正，而不是期待我们一开始就把事情做完美就更好了。
- 我们是一条绳上的蚂蚱。团队中的每一位成员，包括与你们合作的客户，都对需求和这些需求的变化负责。如果这些需求是错误的，你们的错误和客户的错误同样严重，因此抱怨变化是没有意义的。
- 我们不把变化拖到最后。犯错误的确是令人尴尬的事情，但是我们都能意识到这一点，因此要尽早修复错误。这样，才能把损失降到最低。
- 我们不要再把变化当成犯错。考虑到当时掌握的信息，我们已经尽力了，出了错事情才会变得更加明朗，因为开发过程中的各种决策让我们看到现在必须如何改变。
- 我们通过变化学到东西。这是团队成长的最有效方式，也是团队学会更好地合作开发软件的最有效方式。

3.3.3 原则3：频繁交付可工作的软件，从数周到数月，交付周期越短越好

你可能会开始觉得欣然面对变化的想法有点意思，而且可能会对项目有帮助。但是你也许还会觉得这种想法看上去有点可怕。人们对这种想法有这样的反应还是挺常见的。很多在软件团队工作的人，特别是传统项目经理，在第一次听说欣然面对变化的思想的时候都感到很不可思议。这些项目经理每天都要处理各种变化，但是他们面对变化的态度和敏捷方法大不相同。敏捷实践者把对待项目变化的传统态度称作命令－控制（command-and-control）。

“命令 – 控制”这个词来自军事。我们在 2010 年出版的《团队之美》一书中，有一段对 Northrop Grumman（诺斯洛普 · 格鲁门公司，世界第四大军工生产厂商）的首席工程师 Neil Siegel 的采访。Neil 在采访中给出了“命令 – 控制”这个词的军事定义。

> Andrew：我不太熟悉军事系统，“命令 – 控制”系统是什么？
>
> Neil：指的是军事指挥官使用的一种信息系统。通过这个系统，指挥官之间可以互相通信并知晓当前的状况，例如所有人的位置和状态。这是指挥官了解当前情况的方法。过去的战场都不大，指挥官可以站在山头通过望远镜了解当前的状况。但是大约从 1900 年开始，战场开始变得很大，指挥官再也不可能像拿破仑那样站在山头指挥作战了。你开始需要通过技术手段“看清”整个战场。而实现这种功能的系统就称为命令 – 控制系统。

命令 – 控制式项目管理与军事上的命令 – 控制非常类似。

- “指挥”指的是项目经理给团队分配任务的方式。尽管并不是所有成员都直接向项目经理汇报，但是项目经理可以控制所有人的任务分配。项目经理分解工作任务，安排时间计划，然后把任务分配给团队中的人力资源。
- “控制”指的是项目经理管理变化的方式。每一个项目在开发过程中都会遇到变化：工作花的时间比预期的要长，团队成员休病假或离开项目，硬件不可用或损坏，还有其他种种以外的事情都有可能发生。项目经理不断监控这些变化的发生并把控项目：当变化发生的时候对其进行评估，更新项目计划，在进度安排和文档中引入变化带来的改变，给团队分配新的任务，管理利益干系人的期待，不要让人感到意外。

传统项目经理方案对“欣然面对变化”感到不安的原因在于，当她第一次接触到这种想法的时候，她感觉敏捷项目也会遇到传统项目的问题，团队需要响应这些变化。简单地接受变化而且欣然接受变化貌似会在项目中引入混乱。如果敏捷团队不使用命令 – 控制式项目管理，那么它们怎样才能在处理这些变化的同时仍然应对项目团队的日常问题？

欣然接受变化的同时不引入混乱的关键，在于频繁发布可工作的软件。团队通过迭代将项目分割至定期的截止时间。在每一轮迭代中，团队都要发布可工作的软件。在每一轮迭代结束的时候，团队都有一个可以展示给客户的演示，还有一个回顾会议用来回顾本轮迭代的过程以及探讨在本轮迭代中吸取的教训。然后计划下一个迭代要开发什么。可预测的进度安排和持续检查可以帮助团队尽早掌握变化，同时也创建了一个没有责备的氛围。大家在这个环境里可以讨论每一项变化，并制订解决这些变化的方案。

这就是敏捷方法吸引传统命令 – 控制式项目经理的地方。命令 – 控制式项目经理想要控制截止时间。有时间限制的迭代可以实现这一点。此外，这还化解了项目经理的一个最大难题：处理项目晚期发生的变化。传统项目经理工作中最大的困难之一就是监视变化。每日审查和迭代回顾相当于让整个团队帮助项目经理尽早发现变化，从而防止这些变化对项目造成更严重的影响。

项目经理的职责开始远离指挥和控制了。在传统的命令 – 控制方式中，他要做的事情是在团队中部署每日作战计划，以及不断地调整以确保大家都在正轨上。现在，他要与团队一起工作，确保每个人都在关注全局并且朝着同一个目标努力。在发布可工作软件的短迭代中，这些是比较容易做到的。每一位成员都有了具体的目标，而且可以更好地理解大家都

在做什么。此外，每个人都会意识到自己不仅要对自己开发的部分负责，还要为整个团队在迭代结束的时候交付的产品负责。

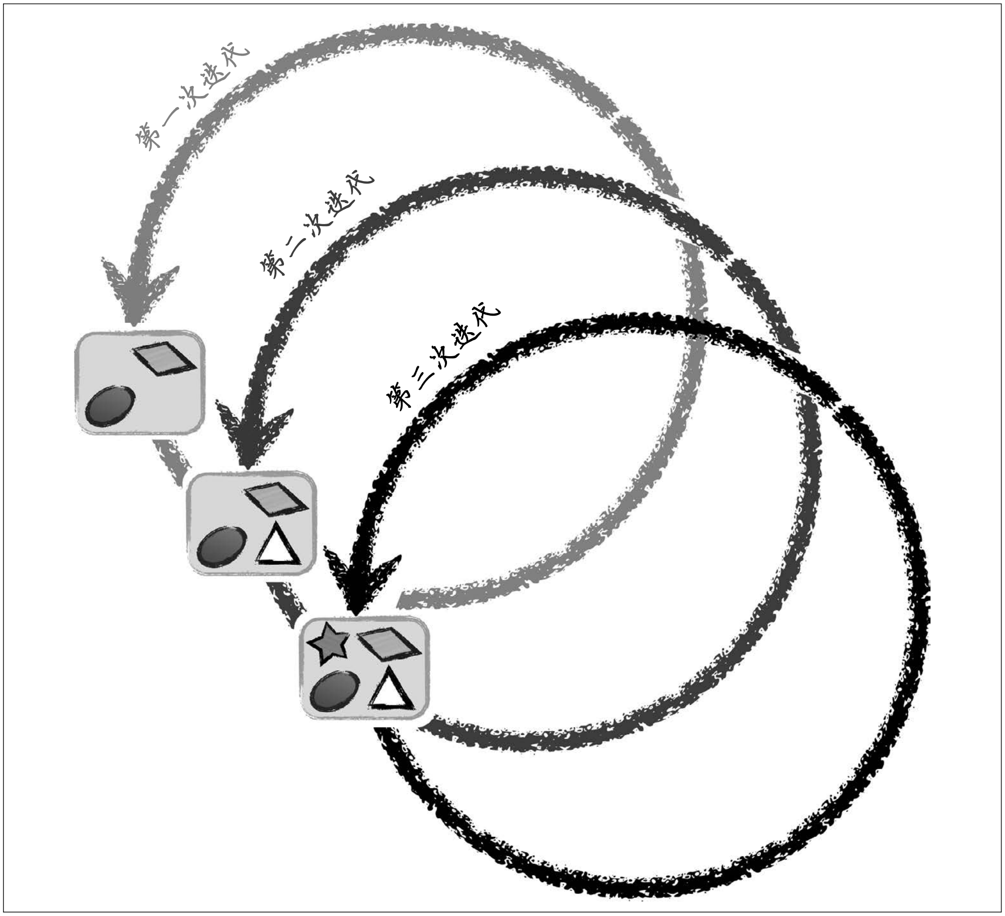


图 3-1：团队通过迭代的方式频繁发布可工作的软件，每一次发布都添加新的特性

3.3.4　改进电子书阅读器团队的项目交付计划

这些原则可以怎样帮助陷入困境的电子书阅读器团队？回想一下这个团队遇到的问题：缺乏竞品都有的重要功能（支持业界标准的电子书格式，允许用户通过电子邮件向设备发送文档），而拥有的功能却没有什么市场（网络书店），他们开发的产品彻底失败了。

假设这个项目重新开始，我们让项目经理和利益干系人一起工作，并且让团队制订以一个月为周期的迭代。这一次项目的进展有很大的不同。

- 在第 3 轮迭代之后，有一位开发人员报告说一种新的电子书格式已经被认定为业界标准。团队决定在第 4 轮迭代中实现一个支持这个格式的库，然后在第 5 轮迭代中把这个支持整合到阅读器的用户界面中。

- 在 10 个月后，团队开发出了一个在原型机中加载并工作的版本，他们可以把这个版本分发给早期公测用户使用。项目经理和这些用户交谈，发现他们非常需要把 Microsoft Word 文档和报刊新闻加载到阅读器中阅读。团队决定在下一轮迭代中将电子邮件功能整合进阅读器，这样读者就可以把文章通过电子邮件发送到设备上。
- 项目进行了一年，利益干系人告诉团队网络书店的功能实际上没有必要开发，因为所有的零售商都会使用标准的电子书格式。幸运的是，这项功能一直在积压工作表的底部，其他迭代周期都在开发更重要的功能，因此这项功能并没有耗费太多时间。

这个团队保证每一轮迭代结束的时候都能交付一份可工作的软件，因此从积压工作表中删除功能意味着他们可以赶早交付了！出版商合作伙伴已经准备好了图书，因为他们的高级经理早就获得了软件的早期版本和原型硬件，都已经试用很久了。出版商一直有参与感，因此他们也有动力尽快准备好要出版的图书，以保证在产品的第一个版本准备好的时候可以让这些书上市。

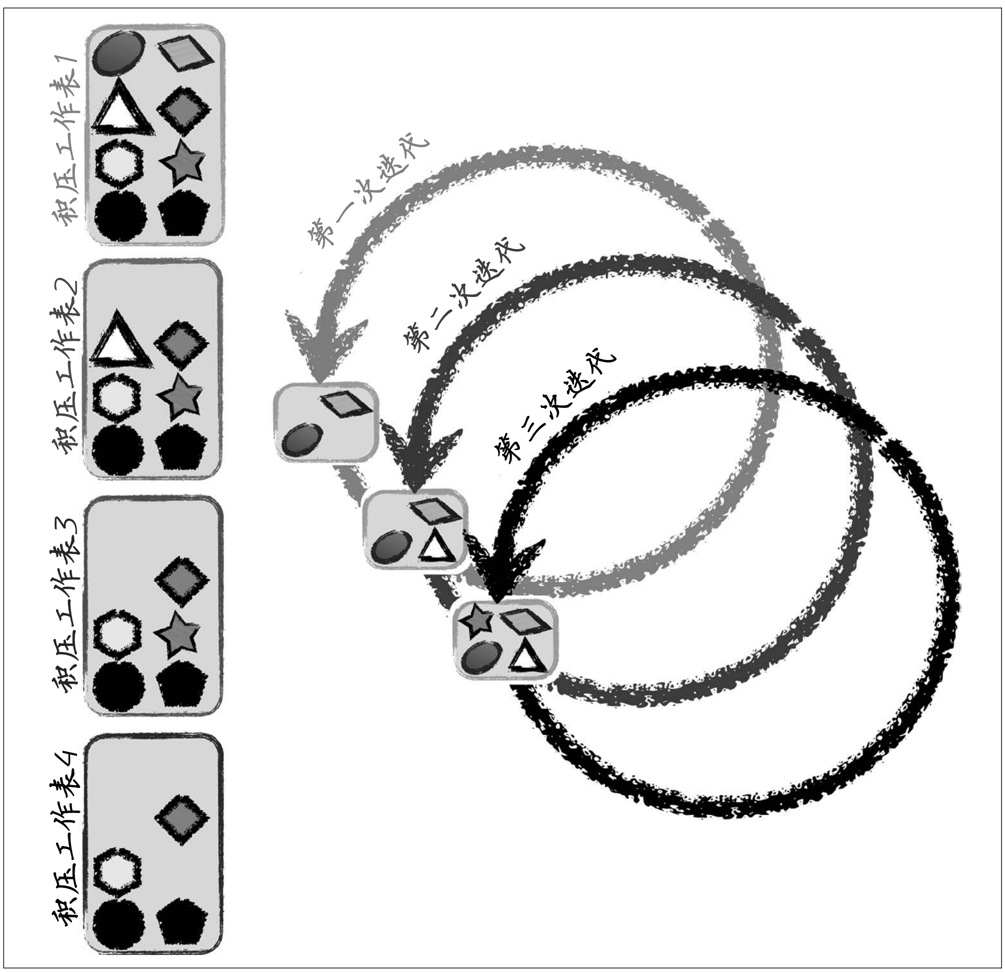


图 3-2：在每一轮迭代开始的时候，团队从积压工作表中选择出要开发的功能

通过持续发布、欣然接受变化以及在每一轮迭代结束的时候发布可工作的软件，电子书阅读器项目的第一款成品的发布大大超前于计划。低效的瀑布式流程一旦定义好需求就把开发团队和客户完全隔离开，而敏捷团队采用的方式完全不同，后者始终与客户交互。这样就可以及时响应变化，开发出更好的产品。

但是这个电子书阅读器团队还远没有达到完美。尽管通过迭代的方式交付可工作的软件，但是大家也被文档压得喘不过气来。所有人都真心感到高兴，因为开发的软件不会卖不出去、陷入困境。但是，每当发现项目确实需要修改的时候，都有一半人返回去更新规格说明书，以保证计划保持最新状态，而且自己的工作在正轨上。看上去，他们更新文档花的工夫与写代码花的工夫差不多。

团队里有人开始讨论如何减少维护这些文档所需的工作量。他们针对文档“正确的详细程度”展开了细致的讨论。但是每当他们想要砍掉一些内容的时候，总会有人指出如果不写下某项功能、需求、设计或测试用例，那么就会有人对其产生误解。如果最终的实现不正确，他们就会因此遭到谴责。于是，文档中的任何一部分看上去都是必要的，因为少了任何一部分团队都有可能开发出错误的软件。

有没有办法可以减少这项负担，同时对项目不产生伤害？还有，一个项目到底有没有“正确”的文档详细程度？

要点回顾

- 敏捷开发宣言随附的 12 条原则让敏捷实践者对实践和方法有了具体的方向和认识。
- 敏捷团队在项目中尽早地获得客户反馈，并且持续发布根据反馈意见改进的软件，从而让客户满意（原则 1）。
- 敏捷团队把变化看作是项目中正面且健康的发展过程，从而拥抱变化（原则 2）。
- 通过设置时间范围的迭代来频繁交付可工作的软件，敏捷团队不断地调整项目，从而给客户带来最大价值（原则 3）。

3.4 沟通和合作

软件团队自开始开发软件以来就一直在纠结“要写多少文档”的问题。很多团队多年来都在寻找解决流程、编程和交付问题的灵丹妙药。与之类似，这些团队也在寻找让文档系统或模板系统记录当前开发软件所需所有信息的神奇方法，并且让这些信息在未来自动维护。

在传统观念中，人们将软件文档看作一种必需的文档管理系统，大家都可以把已有的信息放到这个系统中，然后将自己的信息与其他所有人的信息整合在一起。如果这个系统具有可跟踪性，可以理清楚所有信息之间的所有关系，那么需要的产品、测试，以及部署和维护方式便一目了然。这里的想法是：开发人员可以从设计中的每一个部分追溯到具体的需求，测试人员可以从每一个测试用例追溯到设计、需求和覆盖范围。每当需要变化的时

候，例如需要改变设计中的某个部分，团队就可以准确地看出这样的变更会对哪些代码、需求、范围和测试用例造成影响，因此可以省去很多分析的时间。软件工程师把这个过程称为*影响分析*（impact analysis）。大家通常都会努力维护详尽的可跟踪性矩阵，通过这些矩阵映射范围、需求、设计和测试中所有元素之间的关系。这样，项目经理就可以通过这个映射将代码、bug 报告、设计元素以及需求追溯到源头。

现在回到之前各个团队纠结的问题：到底应该写多少文档？对于敏捷团队来说，答案是够项目开发用就行。具体要写多少文档取决于团队本身的情况，例如团队自身的沟通能力以及要解决的问题等。考虑一下软件开发团队编写的所有其他类型的文档：冗长的会议记录（可以捕捉每一次会议作出的决策）、交叉引用文档（描述了每一项数据或存储）、复杂矩阵（记录每一位成员具体角色、职责和要遵循的规则）。所有类型的文档都遵循一个原则。如果某种文档不能给团队开发软件带来帮助，而且也没有必须写的原因（例如有监管的要求、投资者的要求、高级经理的要求或其他利益干系人的要求），那么敏捷团队就不写这种文档。但是如果发现编写功能性需求文档真的有帮助，那么敏捷团队就可以编写这种文档，这不影响敏捷性。所有这一切的标准就是适合，这也是敏捷给你带来的自由。

至于超出团队开发软件所需要的文档（全面的文档、可跟踪性矩阵、影响分析），消耗在这里的工作很大程度上支持了完整分析任何变更带来的影响。虽然敏捷开发对变化的管理采取了一种不同的方法（通常高效得多），但是敏捷实践者应该意识到传统的文档编写方法实际上与敏捷方法目标一致。在传统的瀑布式项目中，完整文档的全部意义就在于更好地应对变化。

具有讽刺意义的是，完整的文档往往会给管理变化带来阻碍。完美文档和可跟踪性的美梦拥有这样一种系统：团队可以通过这个系统自动地生成任何需要管理的变化带来的影响，可以准确地定位需要修改的地方。

遗憾的是，对于大部分团队来说，通过详尽文档来管理变化的真实情况不这么美妙。团队在项目开始的时候要花大量的时间努力预测未来会发生什么，并完整地记录下来。项目执行的时候，他们需要不停地维护已经写好的文档，并记录所有新开发的内容。如果对于正在开发的产品有了新的理解，他们还需要返回去修订所有受影响的文档。随着时间的推移，过时的文档会越来越多，团队需要花很大的工夫去维护这些过时的文档。

事实上，完整的文档并不完美，所以会导致不必要的变更并浪费精力。文档中任何一部分都是承担特定职责的个人从自己的角度编写的：业务分析师从一个角度写需求，架构师从另一个角度构建设计，QA 工程师从第三个角度编写测试计划。然后他们还要把这些内容整合在一起成为可跟踪性矩阵，结果发现与引入这些割裂视角的期望完全不一致。构建文档本身也成了一个要实现的目标，要求投入的精力越来越多。当变化终于发生的时候，把所有不同视角整合到一个完整文档所耗费的工作全部都需要重做。结果，团队不停地重写文档、重建可跟踪性矩阵、解决冲突。而这些时间本可以用来编写代码、测试软件、部署变化。

得有一种更有效的软件开发方法才行。

3.4.1 原则4：在团队内外，面对面交谈是最有效、也是最高效的沟通方式

敏捷实践者明白，文档只不过是另外一种沟通的形式[4]。写文档交给你的时候，我的目的并不是写文档。目的是确保我脑子里的想法能与你脑子里的想法尽可能地接近。在很多情况下，文档是实现这个目标的好工具，但是文档也不是我们唯一的沟通工具。

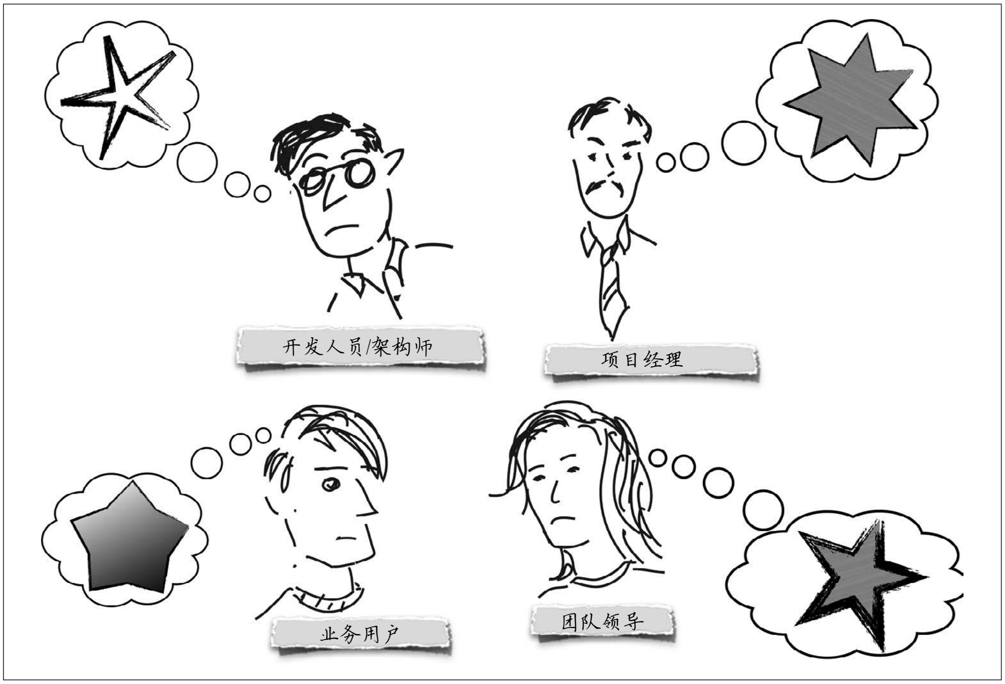


图 3-3：如果团队中的成员不沟通，他们可能会在粗粒度上保持一致，但是最后却朝着不同的目标前进。详尽的文档容易引入歧义，所以更容易产生这种情形

在软件团队中，面对面沟通方式几乎总是优于文档沟通。大家都知道亲自去与别人讨论问题是理解新想法的最有效方法。相比从一页纸或一份 Microsoft Word 文档上读到的信息，我们更善于记住在交谈中传达的信息。因此，敏捷沟通实践更关注个人与个人的沟通，而文档则用来记录那些以后需要回忆具体细节的复杂信息。

幸运的是，对于习惯复杂文档的团队来说，更高效的面对面沟通方式不难解释。这是因为大部分团队并没有尝试达到完整详细文档可跟踪性的理想情况。软件工程师现实得很。发现建立完整文档需要的工作量很大，他们最终会面对面交谈。最后，这也成为他们高效开发软件的唯一方式。由于意识到完整文档通常是不必要的，因而他们也不会因为没编写完

注 4：公平地说，传统项目经理也知道文档只不过是另外一种沟通的形式。一般的项目经理都会研究正式沟通与非正式沟通之间的区别，还会研究书面沟通与口头沟通的区别。他们还会研究沟通中的非语言信息，然后发现面对面沟通是最有效的传播思想的方式。其实在 PMP 考试中就有这些内容！

美的完整文档而愧疚，这本就是不可能做到的事。而事实上，即使他们真的写了这些文档，这些文档在项目中也不会太有用。

团队沟通项目重要想法的最有效方法就是让大家的思考方式保持同步。这样的话，每作出一个决策，大家都可以正确应对。如果一个团队的人用同样的方式去看待世界，而且都能开放地讨论正面和负面的想法，那么大家最终会有一致的视角。如果此时发生了一个变化需要重新思考，团队成员不需要花时间互相解释。

团队沟通的终极目标是形成一种集体意识，在成员之间建立不必直说也能领悟的共同知识，因为反反复复解释同样的事情实在太过低效。如果没有集体意识，团队中不同职责的人需要付出更大的努力才能匹配视角。一个团队越能形成集体意识，越能共享同样的视角，就越容易对同样的问题形成一致的答案。这就为团队构筑了处理变化的坚实基础，可以跳过冲突，立即编写代码，而且不会因为维护文档而分心。

3.4.2 原则5：在整个项目过程中，业务人员和开发人员必须每天都在一起工作

敏捷开发团队有时候会忘记业务人员也有自己的日常工作。在这样的情况下，软件开发团队与他们所服务的业务人员自然会脱节。

为了出色地完成软件开发，开发团队需要与业务人员进行大量面对面的讨论。业务人员了解需要什么软件，因为他们在没有软件的情况下开展了同样的工作，而开发团队可以通过讨论了解这些。因此，对于同软件开发团队一同工作的业务人员来说，软件项目通常只是他们自身工作的一小部分。大部分业务人员希望软件开发团队与他们只有很少的接触。他们只想开一两个会，整理出软件需要完成的功能，就盼着不久之后开发团队带着完美的可工作的软件出现。

而软件开发团队这边则希望尽可能多地与业务人员接触。程序员需要了解软件要解决的业务问题。他们了解的方式就是与业务人员交谈，观察他们工作，分析他们的产出。需要这些信息的程序员恨不得每一个业务人员都可以全天候地解答问题。他们等待解答的时间越长，项目进展就越慢。但是业务人员并不想把自己全部时间都放在软件开发团队上，因为他们自己也有工作要做。那么到底听谁的呢？

敏捷团队有一种绕过这些问题，让业务人员和开发人员双赢的方式。他们首先都认识到，团队要给公司开发带来价值的软件。完成后的软件应该值得公司的投入。如果软件带来的价值超过了开发软件的成本，那么公司就值得在这项开发上投入资金。一个好的项目应该有足够的价值让业务人员感觉到值得投入精力。

当业务人员和开发人员同在团队中开发软件的时候，最有效的方法是让他们在项目完整周期内每天坐在一起工作。原因是，如果换一种做法，业务人员就得等到项目后期再来检查开发团队的工作并给出反馈，而项目后期修改的成本要高得多。如果能尽早处理这些变化，那么大家耗费的时间就会少得多。从整个项目的过程来看，每天与开发团队一起工作，每个业务人员的实际时间开销要少得多。

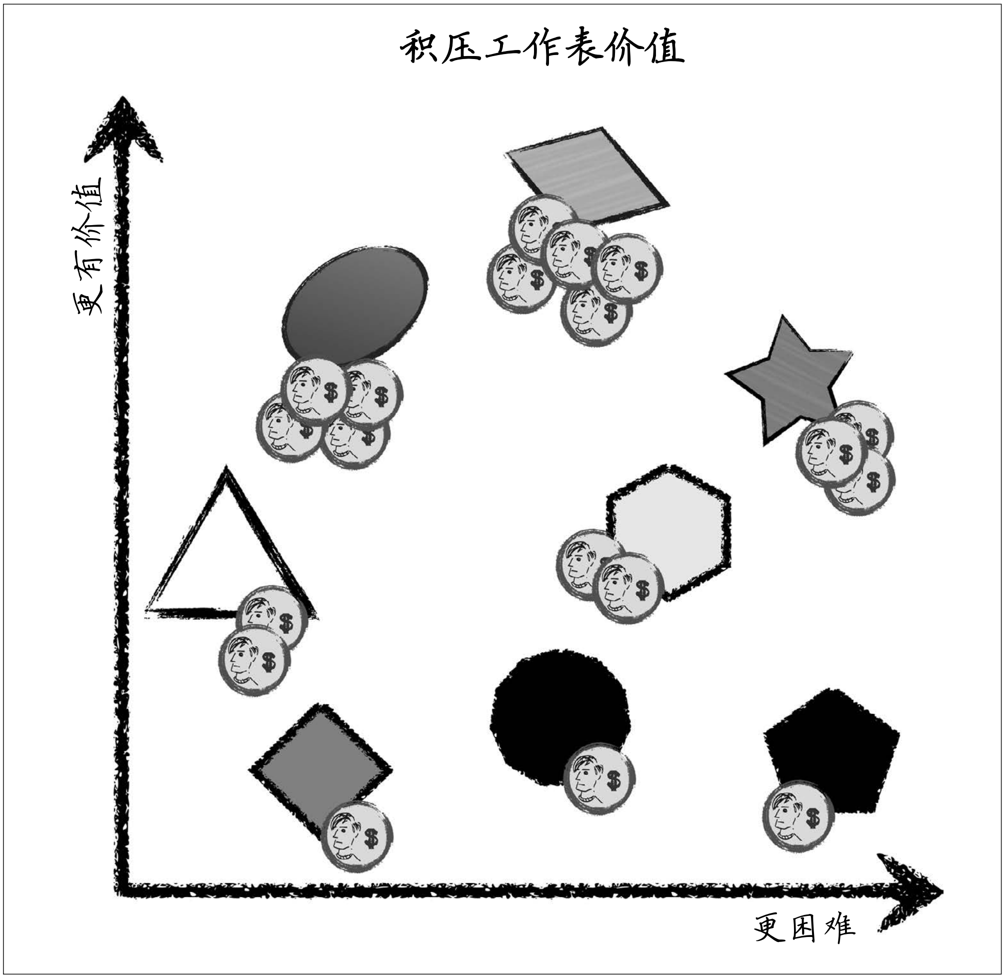


图 3-4：对于公司来说，积压工作表中的一些特性比其他特性更有价值。开发团队在决定每一轮迭代中开发什么特性的时候应该权衡价值（一项特性价值多少）及其成本（开发这项特性要投入多少）

这也是频繁发布可工作软件的开发团队应该把最有价值的特性优先开发的原因，这样的话，业务人员就可以第一时间享用到这些价值。这也是契约的一部分。这也是为什么好的敏捷团队会把业务人员看作是团队中与程序员同等重要的成员的原因。敏捷团队与传统团队存在这样巨大的差异。传统的开发团队把业务用户看作是要谈判的客户；而敏捷团队则是与客户（通常是产品所有者）合作，在项目执行的时候客户具有平等的发言权。（这就印证了“客户协作高于合同谈判”的敏捷核心价值观！）

好的产品所有者会帮助减少业务人员与开发团队在一起的时间。尽管他们仍然每天都要开会，但是产品所有者可以让这个会议关注软件价值的理解以及软件需要解决的业务问题。通过这种方式，团队可以在与业务人员面对面的沟通中验证他们已经从产品所有者那里了解到的信息。

3.4.3 原则6：以受激励的个体为核心构建项目，为他们提供环境和支持，相信他们可以把工作做好

如果公司里每一位同事都意识到团队开发的软件是有价值的，而且团队中所有人（包括产品所有者）都能理解软件是怎样为公司创造价值的，那么项目就可以以最佳状态运行。

相反，如果团队中的成员看不到软件能带来的价值，或是他们没有因为开发优秀软件而得到奖励，那么在这种环境下，项目会失败。常见的情况是，公司绩效审查机制和补偿制度不利于员工采用高效的敏捷方式开发软件，这些做法对项目无益。这背后的问题往往包括以下几点。

- 在代码审查中，如果不断发现 bug，程序员就会获得糟糕的评价，如果没有发现 bug，程序员就会获得奖励。（这会导致程序员在代码审查中不愿意找 bug。）
- 根据发现的 bug 数奖励测试员。（这会导致测试员挑刺并降低报告质量。这种方式给程序员和测试员之间设置了敌对关系，会阻碍测试员和程序员之间的合作。）
- 根据业务分析员产出的文档量判定其绩效评级（而不是根据他们与团队分享的知识量评级）。

最终，所有人的绩效都应该根据团队交付的成果来评定，而不是根据每个人自己的角色来判定。当然，这并不是向领导汇报的时候不要提工作成绩不好或影响团队正常工作的程序员。人们应该根据对整个团队目标所做出的贡献进行考核。但是，考核的方式绝对不能阻碍大家做出超越自己职责边界的贡献。对于一个团队来说，好的环境会奖励下面几种人：认识到软件并没有解决的某个业务问题并将其修复的程序员，以及能够发现代码或架构中的问题并提交给团队的测试员。这样的环境可以给予每一位成员所需的支持，让项目更为成功。

详尽的文档和可跟踪性矩阵可能给团队环境和支持带来潜在问题。这些工作没有在团队中鼓励信任，反而鼓励一心自保（Cover Your Ass，CYA）的态度，在这样氛围下的团队会倾向于采用“合同谈判”的方式，而不是与客户协作的方式。

带着 CYA 态度工作的测试员会努力确保每一项需求都有测试覆盖，而不去考虑测试到底能不能对软件的质量有帮助。带着 CYA 态度工作的开发人员会严格遵循需求文档中的每一个字，而不去认真想一想自己开发的功能是不是真正能给用户带来价值。因为在这种氛围中工作，如果按照用户真正需要的方式进行开发，你可能会被指责不遵守需求规格说明书。业务分析员和产品所有者保护自己的方式就是花时间确保项目范围和需求能整齐地排列在一起。这样的话，他们往往会倾向于忽略一些无法与现有文档保持一致的讨厌需求，而不管这些需求本身的价值。

在把变化看作是坏事的环境中，软件开发团队的成员需要通过 CYA 的态度来保护自己。例如，使用详尽文档的团队很容易排斥变化，因为重新定义项目范围并修改规格说明书、设计、可跟踪性矩阵等工作需要耗费大量的精力。这容易产生内部矛盾，因为在这样的公司中，经理自然会想找一个“始作俑者”为那些因为变化而产生的额外工作负责。当这种责备不可避免时，团队中的成员会逐渐转向编写“防御性文档”以保护自己。这种氛围迫使团队中的每个人都变得 CYA。为了避免糟糕的绩效考核或惩罚，他们可以把责任撇向他们所遵循的那部分文档。

CYA 是信任的对立面。如果做项目的时候只需要编写所需的最少文档，那么公司的氛围就给了团队信任，相信团队在发生变化的时候可以做出正确的事情。在持有同甘共苦态度的敏捷团队中，如果项目失败了，那么大家都需要承担后果。这种团队中不需要 CYA。这样更容易处理变化，因为不需要维护任何不必要的文档。成员可以通过面对面的沟通解决实际问题，仅仅记录下真正必要的内容。他们可以这么做的原因就在于他们知道公司信任他们，即使项目可能耗时超出预期。

3.4.4　在电子书阅读器项目中采用更好的沟通方式

更好的沟通方式一定会让电子书阅读器项目受益。还记不记得那段会议密集的日子？业务分析师仔细地把一大堆详尽的需求整合在一起。这些需求一开始并没有形成万恶的 CYA 氛围。整个团队刚开始真的认为花时间把软件中的每一部分细节都理清楚就可以完整覆盖软件的需求，并且可以开发出最好的产品。他们在一开始的时候就已经把这部分时间用光了，所以在开发的过程中会坚守原始的需求文档，即使发现最终开发出来的产品可能并不适合市场。假使他们当时真的有能力精准地预测两年后市场的需求，那么他们应该能完美地开发出产品。很遗憾，事情发展到最后并没有像当初设想的那样。不过至少没有人被开除，因为大家都可以把矛头指向他们仔细遵循的那份规格说明书。

如果团队一开始就实施更好的沟通方式会怎么样呢？如果团队并不编写完整的需求文档，而只是记录启动项目所需的*最少文档*，那么产品最终会怎么样呢？

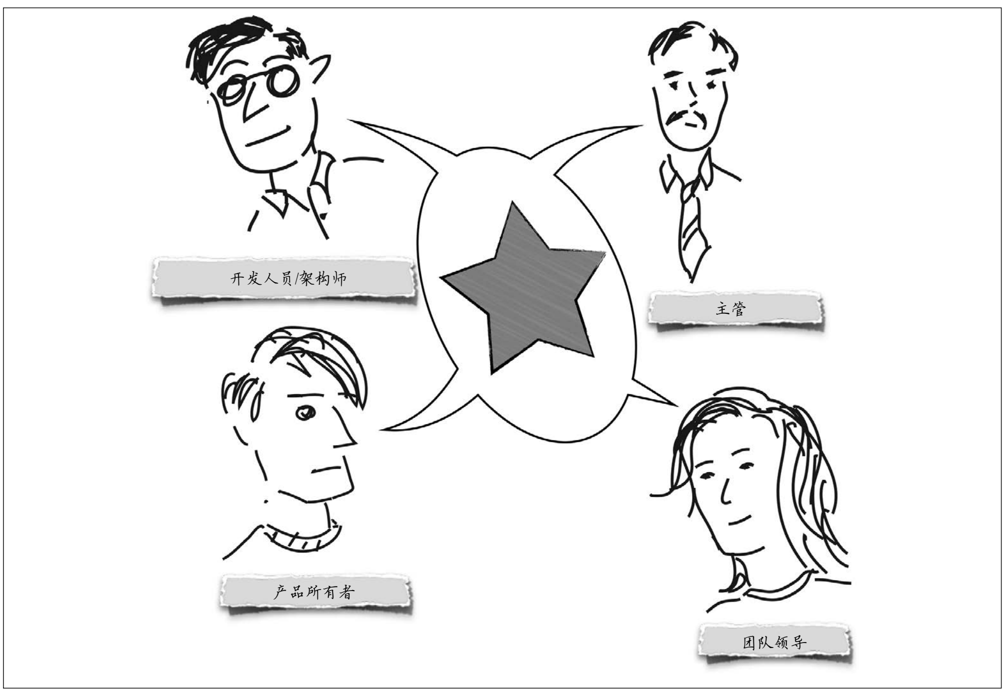


图 3-5：如果团队更多地依赖面对面的沟通，并且使用项目开发所需的最少文档，那么大家就可以更容易地保持状态同步

在整个过程中，大家都要信任其他人可以作出正确的决定。这种做法对于开发电子书格式特别有帮助，因为大家不用绑定在项目刚开始时确定的那种过时格式上，而是可以采用新格式。更好的是，当大家开始开发网络书店的时候，可能会发现这显然不是一个好的想法，废弃这个功能即可。然而，如果列在规格说明书中，这项功能就不可能去掉了。更好的沟通方式会促使项目一直符合最新的需求，从而交付更有价值的产品。

想象一下这对于电子书团队来说有什么不一样。文档负担减少，团队的这部分工作变得轻松。大家发现，更好的沟通方式和文档减负节省了大量时间。但是，如果他们这么做了，而项目仍然不在正轨上，这会是怎么回事？

因为种种原因，他们并没有真正节省这些时间。看上去他们在晚上和周末的加班比以前还多了，因为他们试图在每一轮时间受限的迭代中实现产品所有者承诺的所有功能。似乎他们越敏捷，要做的工作也越多，因为夜晚和周末加班而不能陪家人的时间也越来越多了。这可不是改进！在团队彻底崩溃之前，有没有办法可以阻止这个趋势？

要点回顾

- 过于详尽的文档会增加需求含糊以及团队成员之间误解和沟通不畅的风险。
- 敏捷团队最有效的沟通方式是面对面交谈，并且只依赖项目所需的最少文档（原则 4）。
- 开发人员每天与业务用户一起工作，这样他们可以交付最大的价值（原则 5）。
- 敏捷团队中的每一位成员都对项目有责任感，并且为项目的成功与否负责（原则 6）。

3.5　项目实施——推进项目

有了更好的沟通方式，团队成员相互信任，这是一个非常好的开端。一旦成员相处良好并认准自己在项目中的定位，他们就可以开始解决最重大的问题，在日常工作中做出实事。敏捷团队是怎样推进项目的？

3.5.1　原则7：可工作的软件是衡量进度的首要标准

好的团队合作会确保所有人（包括团队成员、经理、利益干系人以及客户）在任何时刻都了解项目的进展。问题是怎样正确地沟通项目状态——这个问题远比看上去的要复杂得多。

典型的“命令 – 控制”式项目经理会努力地通过详尽的进度安排来保证项目的进度，并通过状态汇报来更新所有项目成员的最新状态。但是状态汇报很难获得项目的真正状态。汇报本身就是一种不完美的沟通工具：也许有三个人读的是完全相同的状态汇报，但他们往往对项目进度持有完全不同的理解。此外，对于项目经理来说，状态汇报也会有极强的政治色彩。几乎所有的项目经理都有过这样的巨大压力：有时候需要在状态报告中略去一些会让经理和团队主管难堪的东西，而别人常常需要用这些信息进行决策。如果状态报告并不够好，进度该怎样汇报呢？

答案就在可工作的软件中。只要真切地看到了软件在眼前工作，那么你就“得到了”项目的进展。你可以看到软件实现了什么，以及没有实现什么。如果经理承诺了要交付的功能没有在软件中，那会很难堪，但是又不可能不去沟通这个问题，因为软件本身就足以说明问题。

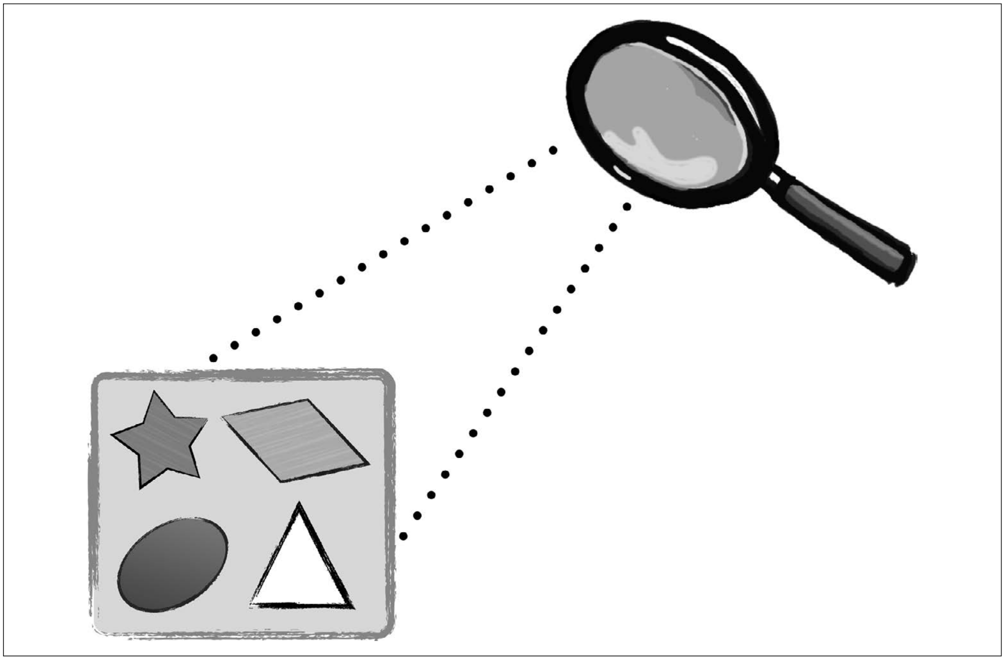


图 3-6：可工作的软件可以更好地给所有人汇报项目最新进展，因为这是团队用来交流当前已经完成工作的最好方式

敏捷团队使用迭代式开发有这样一个理由：在每一轮迭代结束的时候交付可工作的软件，通过真实的产品向大家展示具体成果，团队可以让大家掌握项目进展的最新情况，而且这种方式几乎不可能会让人产生误解。

3.5.2　原则8：敏捷过程倡导可持续开发。赞助商、开发人员和用户要能够共同、长期维持其步调，稳定向前

有很多团队都和电子书阅读器团队一样，疯狂加班，追赶不切实际的截止时间。实际上，硬性的截止时间是命令 – 控制式项目经理采用的主要工具。每当截止时间快要到来的时候，大家的首要选择都是在晚上和周末加班。不切实际的截止时间是一种榨取团队每周额外工作时间的阴险手段。

从长远来看，这种做法是不可靠的。众所周知，一个团队可以拼命工作几个星期干更多的活，但是团队的工作效率一般都会在这段时间过后一落千丈。这是有道理的：人们都会感到疲劳并且失去动力。为了加班，他们推掉了很多重要的公事和私事，然而最终这些事情总是会重新找上门来的。实际上，那些严重加班的团队不会比正常工作的团队交付更多实质工作，而且往往工作质量更糟。

因此，敏捷团队信奉的是维持可持续的开发节奏（sustainable pace）。他们会针对预留的时间制订切实可完成的计划。通过迭代式的开发，这种计划的可行性很高，因为预估接下来两周、四周或六周的工作量远比预估未来一年半的工作量要简单得多。因为只承诺交付能开发的内容，所以团队不会动不动就加班到深夜或周末。[5]

3.5.3 原则9：坚持不懈地追求技术卓越和设计优越，以此增强敏捷的能力

糟糕的预估并不是导致深夜和周末加班的唯一原因。大部分开发人员都经历过这样的事情，一个功能看上去只需要简单编写一点代码，到后面你才发现这实现起来很艰难。这种痛苦郁闷的经历让人耿耿于怀。本来接下来的三个星期可以干别的事情，结果却要用来跟踪各种 bug，给代码打补丁。

从长远来看，避免 bug 比过后修复 bug 要快得多。设计良好的代码维护起来也要简单得多，因为其开发方式易于扩展。

过去二十几年，软件设计领域发生了一次变革。面向对象设计和分析、设计模式、解耦及面向服务的架构等软件设计领域的创新带来了很多模式和工具，让开发人员可以在每一个项目中开发技术优异的软件。

但是这并不意味着敏捷团队在每一个软件项目开始的时候要花大量时间进行大规模的设计。敏捷开发人员会培养起非常好的编程习惯，从而帮助自己编写设计良好的代码。他们不停地寻找设计和代码的问题，一旦发现问题，立即将其修复。在项目的开发过程中，只需要在当下多花那么一点点时间编写可靠的代码并及时修复问题，那么留下的这份代码库在未来就会非常好维护。

3.5.4 改善电子书阅读器团队的工作环境

前面提到的电子书团队成员和家人一定会很喜欢可持续的开发节奏。此外，项目本身也可以进展得更好。这个团队注定从项目一开始就要忍受加班，因为不可能有任何工具制订一年半之后依然精准的计划。

更惨的是，团队的软件设计和架构设计依据都是项目伊始时制订的详尽规格说明书，所以最后出来的代码非常复杂，难以扩展。结果，项目过程中很多小修改都需要打大补丁，代码库中留下了大量的意大利面条式代码。如果团队采用迭代式开发方法，不断交付可工作的软件，那么大家就可以对每一轮迭代制订计划，从而保持一个可持续的开发节奏。通过更简单、更适时的方法设计出来的架构更灵活更可扩展。如果团队使用了更好的设计、架构和编码实践，那么就可以开发出更易维护和扩展的代码。

想象一下，假设采用了这些新的原则，现在电子书阅读器团队就会像一台运转良好的软件生产机器。他们定期地迭代并发布可工作的软件，同时不断地调整方向以确保总是在开发

注 5：这是针对第 1 章讲述的关于“简化”的好例子。此刻，我们只需要告诉你，“可持续的开发节奏”指的就是给予团队足够的开发时间，让成员不需要工作到深夜，也不需要周末加班。本书之后还会进一步深入细节，讨论可持续的开发节奏会对团队的工作环境、公司文化、软件质量产生什么样的影响。

最有价值的软件。他们之间沟通良好，只编写需要的文档。他们用的是很棒的设计实践，开发出了可维护的代码库。而且他们并没有为了做到这些而加班工作。这个团队已经迈入敏捷了！

但是这时下一个项目的阴影已经升起。新的项目经理刚刚给他能找到的所有人发送了大型会议通知。参会者接到消息，相关人等开始订会议室，人们热火朝天地讨论所有需要记录文档的需求。我们刚形成的敏捷团队又开始“心口痛”了。

他们知道马上要发生什么了。大量的需求规范、计划和甘特图会在初期涌现。他们怎样才能保证下一个项目不会掉进前一个项目的坑？他们好不容易才爬出来的。

要点回顾

- 沟通项目进度最有效的方法就是把可工作的软件交付到用户手中（原则 7）。
- 保持团队最高生产效率的方法就是：保持可持续的开发节奏，不要逞强，不要匆忙赶工，避免加班（原则 8）。
- 设计良好并很好实现的软件可以最快交付，因为这种软件修改起来很容易（原则 9）。

3.6 项目和团队的持续改进

KISS（Keep It Simple, Stupid，保持简单）是一项不仅应用于软件开发，也应用于其他所有工程实践中的最基本的设计原则。敏捷团队在项目规划、软件开发以及团队运营的过程中都会遵守这项原则。

3.6.1 原则10：简单是尽最大可能减少不必要工作的艺术，是敏捷的根本

在已有项目中添加代码会让项目变得更复杂，如果还继续添加更多依赖这些代码的代码，项目会变得尤为复杂。系统、对象和服务间的依赖使得变化成为一件困难的事情，因为有了依赖关系，变化导致系统另一部分发生变化的可能性会提升，后面还有可能导致第三个变化。每一个变化都会形成多米诺骨牌效应。采用迭代式开发以及在项目初期将文档量控制到最小，可以帮助你的团队避免交付不必要的软件。

然而，很多开发人员第一次听到诸如“采用迭代式开发”和“以所需的最少计划启动项目”这一类说法时会有一点不适应。他们觉得，如果没有大量的设计和架构决策及其文档化，那么编写项目代码会显得太早，因为如果现在就开始编写代码，那么以后设计变化的时候他们就要删除现在写的这些代码。

这种反应是可以理解的。在软件开发领域之外的其他工程项目中，这种做法并不合情理。比如说，新接触敏捷开发的程序员很有可能会抛出这样的反对意见：“如果让承包商来帮我翻新房屋，我希望可以事先看到完整的蓝图。我不希望他简单跟我聊聊就开始砸我家的墙。”

房屋翻新项目确实不能这样。因为要做的事情是改变一栋房屋，最具破坏性的操作就是拿

大锤子砸墙。但是软件与房屋以及其他物理实体之间是有差异的。在软件项目中，最有破坏性的事情就是编写代码，然后编写更多依赖这些代码的新代码，再然后再编写更多进一步依赖的最新代码。代码的连锁变化会产生可怕的多米诺效应，最终出现一大堆无法维护的意大利面条式代码。

把待定的工作最大化可以避免这种困境，而最好的实现方式就是开发没有太多依赖和不必要代码的系统。而要实现这个目标最有效的方式就是与你的客户以及利益干系人在一起工作，确保只开发最有用且最有价值的软件。如果某一项特性没有价值，那么对于公司来说，更节省成本的方法就是根本不要开发这项特性。为维护这些代码而产生的成本往往比这项特性给公司带来的实际价值要高。编写代码的时候，如果团队可以基于一些只实现单一功能的小型自包含的单元（例如类、模块和服务等）进行设计，那么这个团队就可以避免多米诺骨牌效应。[6]

3.6.2 原则11：最好的架构、需求和设计来自自组织的团队

有大量事前设计的团队非常容易做出过于复杂的设计。想来这并不令人意外。让我们再看一下第 2 章展示的瀑布式流程的示意图，其中有一个完整的阶段是专门用来做需求的，还有一个阶段用来做设计和架构。在设计和架构阶段尽全力就必定意味着要构建可以做到的最棒的架构。对于采用这种方式工作的团队来说，如果提出的需求较少而且设计太简单，那么从直觉上会给人一种偷工减料的感觉。要是他们拿出一份巨大的需求文档和一个复杂的设计，那还会有人质疑吗？当然不会。在流程中有整块整块的阶段让他们做这些事情，意味着人们明确地要求他们这么做。

自组织的团队（self-organizing team）并没有明确的需求和设计环节。自组织团队会用合作的方式对项目进行规划（而不是依赖某个“负责”计划的人），而且会持续地作为一个团队改进计划。采用这种工作方式的团队通常会把项目分解为多个用户故事或其他类型的小块，从能够给公司带来最大价值的块着手，然后再考虑详细的需求、设计和架构。

这种做法使得传统软件架构师的工作更为困难，但是产生的结果却更令人满意。传统的软件架构师坐在一间办公室里抽象地思考要解决的问题。尽管现在很多架构师明显不是这么工作的，但是或多或少与团队日常工作脱节的架构师并不少见。

在敏捷团队中，所有人都对架构负有责任。尽管高级软件架构师和设计师仍然很重要，但是他不再孤立工作。实际上，如果团队从最重要的分块开始逐块构建软件，那么架构师的工作会更有挑战性（而且也更有意思）。敏捷团队不会在一开始就创建一个覆盖所有需求的大设计，而是采用增量式的设计，这就要求设计的系统不仅完整，而且还在项目变化的时候方便团队修改。

3.6.3 原则12：团队定期反思如何提升效率，并依此调整

如果不能持续地改进构建软件的方式，那么团队就不算敏捷。敏捷团队会不断地检查并调

注 6：本章在这里又简化了，没有深入进去。我们之后会更详细地讨论团队怎样在项目初期不做大量设计的情况下开发优秀的代码，讨论这种做法对项目的影响，以及如何包容项目未来的变化。

整，成员会检查自己项目运转的方式，并通过检查的结果对未来进行改进。而且，他们不只是在项目结束的时候这么做。他们会每天开会寻找需要改变的地方，如果有道理，就会改变当前的工作方式[7]。你需要适应的一点是：对于你自己以及你团队中的其他同事，你必须非常诚实，让大家知道哪些事情可行，哪些不可行。对于刚刚迈上敏捷开发道路的团队来说，这一点尤其重要。增强团队实力的唯一方法就是经常回顾自己已经做的事情，然后评估作为一个团队这些事情做得怎么样，最后提出能改进的计划。

这是有道理的，而且几乎所有人都同意这件事情有意义。回顾过去，清点哪些事情做对了，哪些事情做错了，这确实是很多团队真心想做的事情，但是很少落实。原因之一就是，这一开始并不让人感到舒服。这需要揪出具体的问题和错误，而很少有人会对公然指出其他人的错误感到自在。随着时间的推移，团队中的成员会对这件事情感到越来越自然。最终，大家会认为这是提意见而不是挑刺。

很多团队不反省的另一个原因是没时间。即使找到了时间，人们也常常觉得投入下一个项目比思考已经完成的事情要更重要。如果在开始每一个项目的时候，给每一轮迭代以及每一个项目的收尾阶段预留了开会时间，总结并评估已经完成的事情，而且制订出改进计划，那么团队更有可能真正地坐在一起讨论他们已经完成的事情。这样可以从经验中吸取教训，提高效率。

要点回顾

- 敏捷团队避免开发不必要的特性以及过于复杂的软件，保证给出的解决方案尽可能简单（原则 10）。
- 自组织的团队对项目的所有方面都负有共同的责任：从产品构思到项目管理到项目的设计和实现（原则 11）。
- 敏捷团队在每一轮迭代结束的时候和项目结束的时候会花时间总结过去，讨论总结经验，提高开发软件的能力（原则 12）。

3.7 敏捷项目：整合所有原则

敏捷开发在软件工程的历史上是独一无二的。敏捷开发与多年来“银弹”方法的浪潮不同，后者承诺通过神奇实践、高级软件工具，以及昂贵的咨询费用解决软件团队中的问题。

收获“聊胜于无”结果的团队和通过敏捷开发享受到更多好处的团队有一个区别：后者意识到不能像点菜一样选择敏捷实践。组合使用这些实践的关键在于团队的思维方式，敏捷价值观和原则是思维背后的动力。

敏捷的独特之处在于从价值观和原则出发。敏捷团队不仅要诚实地回顾开发软件的方式，还要回顾成员交流的方式，以及与公司其他同事交流的方式。首先要理解原则，然后再采用方法，要完整理解其工作原理，还要在过程中不断地评估和改进。敏捷团队可以真正找

注 7：这里又是一个简化。目前，我们只需要了解敏捷团队会自查和改进即可。第 4 章我们会学习 Scrum 团队如何具体做这件事情，以及这件事情和自组织的团队有什么关系。

到改进项目运行的方法，增强敏捷性，开发并交付更好的软件。

常见问题

我是一名“明星”开发人员，我只想让其他人都不要打扰我，这样我就可以开发出伟大的软件！我为什么还要考虑任务板或燃尽图这样的东西？

所有优秀开发人员都遇到过这种沮丧的事情：本来开发了一段很棒的代码，结果某个根本不知道怎么编程的家伙要求做出一个修改，你就不得不把这段代码弄乱然后打上补丁。对于非常在乎程序设计技艺的人来说，如果一些非开发人员故意等项目做到一半才想明白到底需要什么，那么进行的技术妥协（而不是一开始就编写正确的代码）就是不必要而令人沮丧的。

这也是很多伟大开发人员被敏捷团队吸引的原因。没错，敏捷开发要求你关心计划，习惯使用诸如任务板和燃尽图之类的计划工具。敏捷方法构筑的根基在于这些与计划相关的实践，而且这些实践是专门挑选出来的，已经简化到了最极致，并且足够满足高效计划和运行项目的需求。计划项目的最大好处就是可以在规划期过问棘手问题，防止收尾时进行变更。因为高效的敏捷团队从项目一开始就在不停地与用户沟通，所以可以做到这一点。很多优秀的开发人员都会赞同在一开始做计划的时候问用户一些棘手的问题，而这些问题如果不问的话可能会在项目后期引发变化。避免最后才出现的需求变更，避免打乱已经编写好的代码打补丁，这也是敏捷开发很好的卖点。

敏捷计划还意味着与团队其他成员沟通，这种沟通可以真正帮助伟大的开发人员进步。“明星”开发人员从不停止学习，但是在命令－控制式的团队中，明星开发人员与团队的其他成员隔离开了，他们的大部分学习都是自我主导的。而在自组织的团队中，开发人员之间会进行大量的沟通，这种沟通指的并不是被迫参加无穷无尽的无用状态汇报会。团队成员自己决定为使项目正常运转而沟通的内容。这样不仅可以做出更好的项目，还意味着你可以向坐在身边的开发人员取长补短。比如说，坐在身边的同事要在工作中采用一种你之前没有见过的新设计模式，那么在项目结束的时候，你就会了解到这个新设计模式好不好。如果好，你就可以把这个新技术添加到自己的工具箱中。这种学习过程是自动发生的，你不需要费额外的力气，因为整个团队都在高效地沟通。这也是接纳敏捷的开发人员往往发现自己技术更强的原因之一，他们感觉自己的编程技艺在不断地进步。

我是一名项目经理，我现在还不清楚我可以怎样融入一个敏捷团队。我在这里面的职责应该是怎样的？

如果你是一名项目经理，那么你的职责可能属于以下三种传统的项目管理职责之一。

- 一名杂事缠身的计划者：需要预估并指定项目进度，还要指导团队的日常工作。
- 一名产品专家，可能承担了业务分析师的职责，需要确定需求、与团队沟通需求并确保团队开发出满足需求的软件。

- 与高级经理以及与公司高层管理人员一起工作的主管，需要向他们汇报公司在项目上的投资正在产生价值。

在第 4 章中，你会进一步学习一项最常见的敏捷技术：Scrum，还会学习 Scrum 团队中的各种职责。如果你是一名会深入团队细节的实干型项目经理，那么你就很适合去做 Scrum 主管。Scrum 主管的工作是帮助团队制订计划，并且在整个项目的过程中帮助团队成员排除影响进度的困难，让团队最终能交付软件。换一个角度，如果你的工作是理解公司的需求并负责向团队沟通需求，那么你更有可能成为产品所有者。在这种情况下，你要负责管理积压工作表，决定进入每一轮迭代的特性，并且在整个项目的过程中负责回答团队提出的细节问题，确保团队可以保持在正轨上，开发出正确的软件。

如果你是一名负责管理的项目经理，那么你带领的团队很有可能不敏捷，不过也没关系。你反而会要承担一名敏捷战士的最重要职责，在你的团队和经理中推行敏捷实践和敏捷价值观。如果有一些分解为多轮迭代的特性积压工作表，并且掌握了本轮迭代相关细节，你会发现你恰好掌握了向高层管理人员和高级经理汇报所需的具体信息。越清楚进展以及目标完成程度，你就越了解项目的真实进度。但是为了真正做到这一点，你还需要熟悉敏捷团队的工作方式和交流方式，让团队和领导层之间沟通顺畅。

稍等一下。如果整个团队在一起做计划，那么这是不是说没有人负责这件事情？这看上去很不实际。决策是怎样作出的呢？

这取决于要做什么决策。解决冲突的方法应该是不变的。考虑一下你现在所在的团队，或回想一下你之前所在的团队。谁在负责？谁负责解决团队成员不能解决的争辩和分歧？谁考核你的绩效？公司的层次结构有很多种，而敏捷团队可以在任何层次结构下工作。然而，敏捷团队更擅长自己解决冲突，因为关注沟通，而且其目标比其他类型的团队更容易达到一致。

如果你问的是哪些特性会进入软件，或者通过哪些方法来开发这些特性，那么这些事情是敏捷团队中特定角色负责决策的。在 Scrum 团队中，产品所有者负责决定软件中需要实现哪些特性。然而，团队只需要接受那些依据真实信息估算可以塞进一轮迭代的特性。第 4 章会进一步讨论如何计划。对于敏捷团队来说，计划是属于整个团队的，因为这种团队是自组织的。

但是，计划属于整个团队并不意味着团队群龙无首，负责人还是有的。如果你们刚刚开始步入敏捷，你一年后的老大很可能就是你现在的老大，区别在于，你的老大会对敏捷足够信任，从而让团队拥有项目决策权，而且会支持团队的决定，而不会事事插手或猜疑。这也是敏捷在真实世界中可以行得通的唯一方式。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 如果你现在正在开发一个项目，那么在开始编写代码之前首先与团队坐在一起，花 15 分钟讨论一下你们正在开发的特性。你能不能发现对于同一项特性的分歧？
- 列出你现在正在开发的特性。尝试通过价值和开发难度的维度组织这些特性。
- 花几分钟列出你和你的团队会创建和使用的所有文档。看看能不能找出一些团队开发代码实际上并不需要的文档？
- 下一次加班到很晚的时候，思考一下加班的原因是什么。能不能想出一些做法，让你和你的团队避免加班？截止时间是不是太勉强了？是不是在最后一刻加进了额外的工作？首先要明白加班是有问题的，然后花时间去了解导致加班的原因是什么，这是解决问题的第一个步骤。

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 《敏捷软件开发（原书第 2 版）》，Alistair Cockburn 著：进一步了解敏捷开发宣言的价值观和原则，以及宣言创建的过程。
- 《敏捷项目管理（第 2 版）》，Jim Highsmith 著：进一步了解敏捷项目管理的价值观、迭代和其他方面。
- *Succeeding with Agile*，Mike Cohn 著：进一步了解团队在步入敏捷时会遇到的困难以及克服这些困难的方法。
- 《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》，Lyssa Adkins 著：进一步了解如何避免命令 – 控制式思维方式。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 帮助团队意识到超长时间的工作并不能带来更多的代码。实际上这样开发的代码量会更少，而且质量也会受影响。
- 与团队成员单独坐在一起讨论工作。哪些事情激励了成员？哪些事情让他们感到沮丧？他们作出决策的依据是什么？
- 让每一位团队成员选出对自已影响最大的三条敏捷原则，既可以是正面影响也可以是负面影响。人们会很惊讶地发现在团队中的其他同事选了不同的原则。这有助于帮助大家找到共同点。
- 以大家共有的原则作为起点，找到最适合团队思维方式的一组实践。

第 4 章

Scrum和自组织团队

无法用于实践的大道理只不过是过眼烟云。
而没有指导原则的具体实践往往会被误用。

——Jim Highsmith[1]

棋牌类游戏 Othello 有一个口号：“一分钟就可以学会，但要用一辈子时间去精通。”这句话非常适合正在学习 Scrum 的团队。Scrum 的基本实践和原理非常简单，采用起来并不困难。但是要深刻理解 Scrum 价值观并且通过这些实践和原理开发出更好的软件则是一件更富有挑战的事情。

Scrum 的规则很简单，也很容易讲明白。对于很多要采用敏捷的团队，这是一个非常好的起点。下面是 Scrum 项目的基本模式。

- 在 Scrum 项目中有 3 种主要的角色：*产品所有者*、*Scrum 主管*和*团队成员*。
- 产品所有者和团队其他成员一起工作，负责维护生产积压工作表（production backlog），并对表中的项制订优先级。
- 软件在多轮时间限定的迭代中完成开发，这些迭代称为*冲刺*。在每一轮冲刺开始的时候，团队进行*冲刺规划*，从积压工作表中选择出这一轮要开发的特性。确定的列表称为*冲刺积压工作表*，团队利用完整冲刺的时间完成这个列表中所有特性的开发。
- 团队每天碰面，开一个短会，更新成员各自的进度，并讨论遇到的困难。这个会称为*每日 Scrum 会议*（Daily Scrum）。每个人都要回答 3 个问题：自上一次每日 Scrum 会议以来，我都干了些什么？从现在起到下一次每日 Scrum 会议的时间内我要做什么？我遇到了什么困难？
- 有一个人（Scrum 主管）要和整个团队一同工作，帮助团队成员克服困难，保证项目正

注 1：《敏捷项目管理（第 2 版）》，Jim Highsmith 著。

常运转。在每一轮冲刺结束的时候，会有一次*冲刺评审*（sprint review）向产品所有者和其他利益干系人展示可工作的软件。团队还会召开*回顾会议*，找出要从这一轮冲刺中吸取的经验教训，这样就可以在未来改进开发软件的方式。

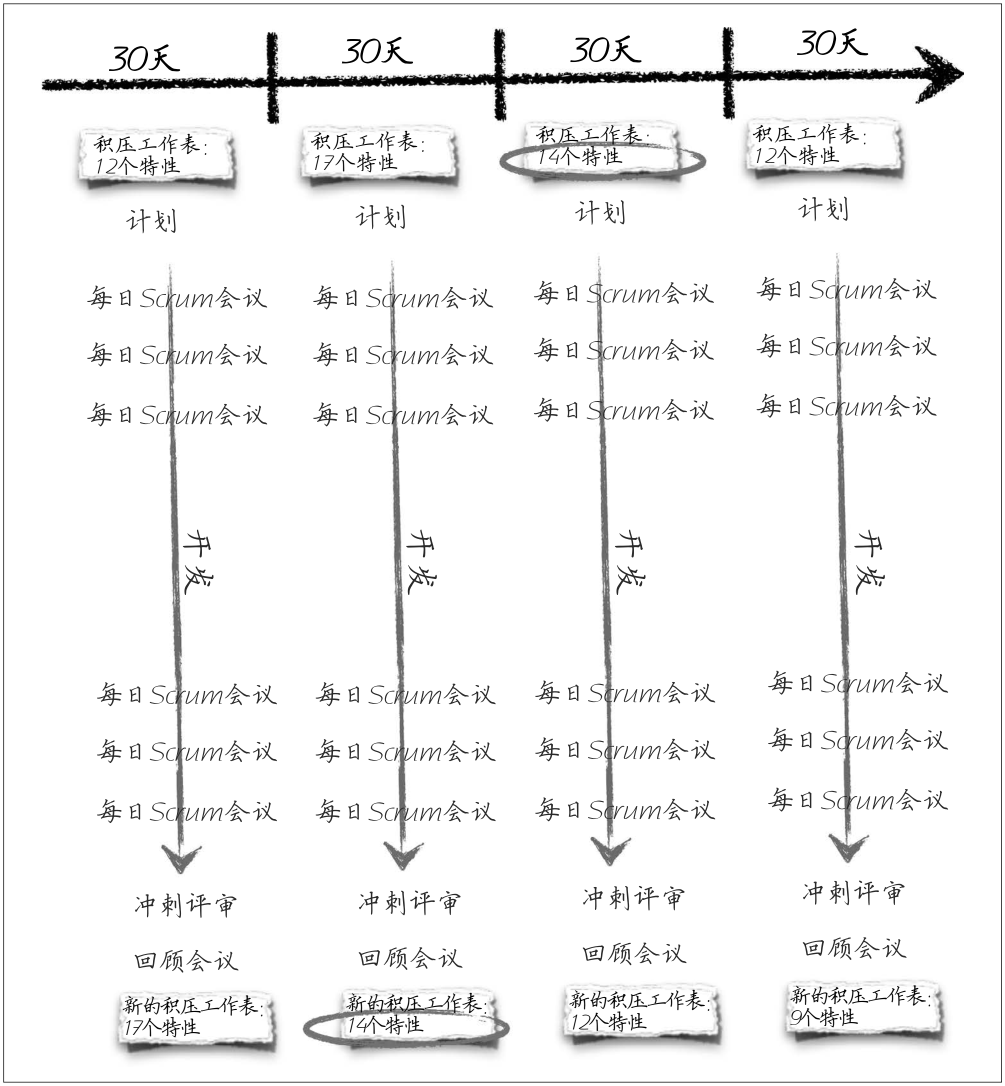


图 4-1：基本 Scrum 模式

但是对于 Scrum 团队来说，想要变得高效，仅仅遵循这些基本的模式还不够。高效 Scrum 团队是*自组织*的，Ken Schwaber 在《Scrum 敏捷项目管理》一书中有如下陈述（注意那些表示强调的黑体词）：

> 为了让 Scrum 发挥作用，团队必须深刻理解**集体承诺**（collective commitment）和**自组织**。Scrum 的理论、实践和规则都很容易掌握。但是如果一组成员不能在固定时间内完

成交付，实现集体承诺，**那他们就没有真正实现 Scrum**。如果团队成员不再像零散的个人，而是有一个共同的目标，那么这个团队就有能力实现自组织，并能快速解决复杂问题，产生可执行的计划。

这一章的目标就是通过第 2 章和第 3 章建立起的思想，向你讲解 Scrum 的实践和模式，让你真正实现 Scrum。我们会通过这些实践来展示集体承诺和自组织原则背后的思想。

4.1 Scrum的规则

在《Scrum 敏捷项目管理》一书中，Ken Schwaber 列出了 Scrum 的规则，描述了 Scrum 项目的基本模式。你可以在 www.scrum.org 网站上下载 Scrum 定义性规则的免费 PDF，这份 PDF 文档是由 Ken Schwaber 和 Jeff Sutherland 编写的名为 *The Scrum Guide* 的电子书。他们创建了 Scrum 方法，并将其推广到业界。看过了第 2 章和第 3 章的介绍，现在你应该很熟悉这些规则了。一个典型的 Scrum 项目遵循以下规则。

- 每一个冲刺开始的时候，Scrum 主管、产品所有者和团队其他成员在一起开会制订计划。这个会议分为两个部分，每一个部分限定时间为 4 小时。在冲刺计划之前，产品所有者需要做功课，提前准备好一份按照优先级排序的产品积压工作表，其中列出了用户和利益干系人要求实现的功能。在会议的前半部分，产品所有者和团队一起选择出要在这一轮冲刺结束时交付的特性，选择的依据是这些特性的价值以及团队对工作量的预估。团队同意在冲刺结束的时候演示一份包含这些功能的可工作的软件。会议前半部分有时间限制（30 天冲刺，则会议限时 4 小时，短一点的冲刺，会议时限可以按比例缩短），在结束的时候，不管讨论了多少内容，团队都要定下这一轮冲刺的积压工作表。在会议后半部分，团队成员在产品所有者的帮助下整理出实现这些功能要用到的具体任务。这一部分的会议也是根据冲刺的长度来限定时间的，但是通常比会议前半部分时间短。在冲刺规划结束时，冲刺积压工作表要拟定。
- 团队每天召开 Scrum 会议。所有的团队成员（包括 Scrum 主管和产品所有者）都必须参加[2]，其他感兴趣的利益干系人也可以参加（但是必须保持安静，只能作为观察者）。这种会议时间限定为 15 分钟，因此所有的团队成员都必须准时出席。每一位团队成员回答 3 个问题：自上一次每日 Scrum 会议以来，我都干了些什么？从现在起到下一次每日 Scrum 会议的时间内我要做什么？我遇到了什么障碍和困难？每一位团队成员的回答都必须简练。如果需要讨论，那么相关人员可以安排会后立即讨论。
- 冲刺要限定在规划的时限内：很多团队选择 30 个自然日，不过这个时间长度也可以调整，有些团队选择两周冲刺，还有一些团队选择一个月冲刺（规划会议的时间限定也应该作出相应调整）。在冲刺中，团队将冲刺积压工作表中的特性开发到可工作的软件中。他们可以从团队外获得帮助，但是外人不可以指挥团队成员，而且这些人必须值得信任，他们一定要按时交付软件。如果团队中任何人在冲刺的过程中发现他们的承诺超过了实际能力或者可以增加更多的项目，那么他们必须确保第一时间通知产品所有者。产品所

注 2：对于你的团队来说，每天开会是不是不现实？也许成员分散在不同的团队中，或是还有别的工作牵扯精力？你是不是已经开始寻找替代每日 Scrum 会议的方式了？例如替换为在线会议或 wiki 页面？这些迹象表明，你的思维模式需要变化，否则无法从 Scrum 获得最佳成效。

有者可以与用户以及利益干系人一起工作，并且调整他们对项目的预期，让冲刺积压工作表与团队的真实能力匹配。如果发现冲刺还没有结束，团队就已经没活可干了，那么他们可以进一步充实冲刺积压工作表。团队必须保证冲刺积压工作表时时更新，而且所有人都能看到。在异常或极端情况下（例如出现了严重的技术问题，发生了组织变动或人事变动），如果团队自知无法交付可工作的软件，产品所有者可以提前终止冲刺并发起新一轮的冲刺规划。但是大家都必须认识到终止冲刺应该是非常罕见的事情，这会极大地阻碍团队开发和交付软件，而且会严重地损害他们与用户以及利益干系人之间的信任。

- 在冲刺结束的时候，团队召开冲刺评审会，向用户和利益干系人展示可工作的软件。演示只包含那些真正完成的功能[3]（在这个例子中，完成意味着团队已经做完了这项特性所有的相关工作，包括测试，而且产品所有者已经确认功能完成）。团队只能展示可工作的软件，而不包括一些中间产物，例如架构图、数据库模式以及功能说明书等。利益干系人可以问问题，团队可以回答问题。在演示结束的时候，团队会询问利益干系人的意见和反馈，后者也有机会分享想法、感受和观点。如果需要，下一轮冲刺规划可以考虑加入修改。产品所有者可以把修改的内容添加到产品积压工作表中。如果需要立即修改，那么这些修改的需求就会出现在下一轮冲刺的积压工作表中。
- 在这一轮冲刺后，团队会召开一个冲刺回顾会议，讨论可以改进工作方式的方法。团队和 Scrum 主管参会（产品所有者也可以参加）。每个人都要回答两个问题：在这一轮冲刺中有哪些事做得不错？未来有哪些事情可以改进？ Scrum 主管会记录所有可以进行的改进，然后以非产品项充实产品积压工作表，例如设置新的构建服务器，采用新的编程实践，以及改变办公室布局等。

就是这样，很简单！

好吧，也许这并不简单。如果真的这么简单，那 Scrum 怎么还没有遍地开花？为什么有些采用了 Scrum 并遵循所有规则的团队感觉只收获了“聊胜于无”的结果？我们错过了什么？

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试采用敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

4.2 第1幕：Scrum的适用条件

Hover Puppy 是一家开发网站和手机应用的小型软件公司，已经连续几年有不错的业绩了。半年前，他们最新的一款手机应用销量很不错，因此 CEO 决定把挣到的钱投入到一个新

注 3：“完成”的意思很明确，即真正完全做完，不需要任何其他工作。但是实际上这很微妙。这又是一个简化的例子，我们在本书中还会多次提到这个概念。把你的眼睛瞪大了！现在你已经看到过好几处这种类型的简化了，后面还有，但我们不会再通过脚注提示你了。

的项目中：一个名为 Lolleaderz.com 的网站，用户可以在这个网站上为宠物视频创建积分榜和成就。

项目刚启动的时候，团队主管 Roger 想要采用敏捷开发。发现团队和他一样对前景感到兴奋，他真心感到高兴。手机应用团队是同一公司的另一个团队，他们已经通过 Scrum 开发了大为成功的手机应用，他们的思想已经在公司广为流传。Roger 团队里有人开始称 Roger 为 Scrum 主管，Roger 承担了这个角色。

Roger 做的第一件事情就是寻找产品所有者，但是找到一个合适的产品所有者并不是很简单的事情。幸运的是，Hover Huppy 是一家所有人都可以对 CEO 称名道姓的小公司。Roger 跑去找 CEO，并对 CEO 解释了一下现状，将产品所有者描述为这个项目的“利益干系人之王”。此时，有一个客户经理 Avi 刚好完成了一个项目。CEO 就把 Avi 介绍给了 Roger，并且向他们表示完全赞同使用 Scrum，让他们自己考虑怎样把 Scrum 用起来。

项目一开始看上去进展得非常顺利。Roger 设定了长度为一个月的冲刺。在每一轮冲刺开始的时候，Avi 都准备好一个要构建的特性的积压工作表。Roger 召集每日 Scrum 会议。而 Avi 在自己的日程表中预留了时间，他每天都可以出席会议。第一轮冲刺进行得很不错，大家都相处得很好，整个团队在取得进展。在这一轮冲刺结束的时候，团队向 Avi 演示了网站的一个简单版本，其中实现了他们计划开发的几个功能。看上去他们对 Scrum 的尝试还挺成功。

可是在之后的几周里，项目中开始出现分歧，不过一切看上去还在正轨上。有一个客户经理的客户是一家电影出品公司，这家公司想要把暑期档大片的广告放在 Hover Puppy 的每一个网站上。团队承诺 Avi 他们可以在冲刺结束的时候演示这项功能。但是他们早些时候遇到了一些技术难题，这项功能被推到下一轮冲刺中了。Roger 解释说，Scrum 团队交付的软件一定是可工作的，因此在一轮冲刺结束的时候如果有一项功能没有完成，这项功能就会被推到下一轮冲刺中。但是他们对此也不是十分确定。

随着一轮轮的冲刺迭代，项目的进展似乎放缓了。在第 3 轮冲刺结束的时候，Avi 感觉他和团队工作的时间越来越长，而他自己和客户一起开展本职工作时间越来越少。在项目初期，他以为他能掌控团队的工作。现在他开始觉得被选为 Lolleaderz.com 网站项目负责人是一件倒霉的事情。Roger 听到 Avi 和其他客户经理抱怨他的团队很难合作，他很沮丧。

更糟糕的是，整个团队开始对这一堆 Scrum 事务感到厌烦了。Roger 和 Avi 还没找到时机解决这个问题，团队里就开始出现另一个小危机。有三名开发人员跑来找 Roger 谈话，抱怨新的每日 Scrum 例会。有一个人说：“我每天上班要干的活就够多了，这些会就是在浪费我们的时间。我为什么要在这里听完整个每日进度汇报，然后等你给大家分配下一天的任务？给我发电子邮件不就行了吗？”Roger 没法完美地解答这个问题。他说这就是 Scrum 的规则，开发人员坚持参会就可以了。但是没有人对这样的答案感到满意，Roger 也开始怀疑开发人员是不是也有道理。

当 Roger 开始计划第 4 轮的冲刺时，大战爆发了。Avi 坚持说让用户给视频“点赞”评分的功能必须进入这一轮冲刺，否则就会流失广告。但是这项功能涉及大量数据库相关工作，还需要骨干级数据库管理员（Database Administrator，DBA）对数据模型做很大的修改。这意味着目前他们没有时间编写存储过程。看上去他们这项功能需要推迟一周才能做

好，但是推到下一轮冲刺也是不可接受的。

现在已经过去了 6 个月，项目经历了 5 轮冲刺。Roger 感觉 Avi 对团队提出的要求越来越多。Avi 感觉沮丧，因为这个项目并没有交付出一个可以卖给客户的网站。在两轮冲刺之前，团队就应该开发完成视频标签页面和社交媒体 feed 功能，但是这些现在还没有交付。在最近一次客户经理会议上，Avi 责怪团队推迟了进度。Roger 了解到这个项目有被取消的风险。

Roger 更是忍无可忍。项目一开始进展非常好，后来却成了一个怪物。他感觉自己有责任，但是不知道如何修复这些问题。他回顾了用来学习 Scrum 的各种书籍和网站。从读过的各种资料上看，他做的所有事情都是正确的，至少书上是这么描述的。他做到了冲刺，开了每日 Scrum 例会，也开了回顾会议。他和产品所有者一起确定积压工作表的优先级，在每一轮冲刺中从积压工作表中找出最有价值的特性开发，并且与团队一起估算是否能完成，然后把任务指派给开发人员。

Roger 打足精神给团队鼓劲："我刚才被 CEO 叫到办公室谈话。我们项目进度拖延，这让他很不高兴。Avi 也不高兴。你们看，我会保护你们，承担责任。但是我们需要继续整理预估，因为项目进度已经远远地落下了。我真的需要你们弥补这个错误，希望你们可以周末加班来赶上进度。"

这是过去两个月里他第三次要求大家晚上和周末加班了。这个项目看上去好像和过去那些失控的项目一样。

那么是哪里出问题了呢？你能找到问题的根源吗？如果被指派为这个项目的 Scrum 主管，你会怎么做？考虑一下敏捷价值观和敏捷原则。有没有办法应用这些价值观和原则，让这个项目运转得更好呢？

4.3 Scrum团队中每个人都要对项目负责

每一个 Scrum 项目都有一位产品所有者、一位 Scrum 主管以及一个团队。但并不是说有这些角色的项目就是 Scrum 项目。Scrum 中的产品所有者与典型瀑布式项目中的产品所有者不同，后者只管"事先确定好全部需求"。Scrum 主管也不是发号施令、控制一切的项目经理（即技术团队主管）。当 Scrum 主管、产品所有者和团队一起工作而不是分开工作时，这个项目才开始有 Scrum 的味道。

4.3.1 Scrum主管指导团队的决策

主管的工作方式不同是 Scrum 团队与传统命令 – 控制式项目团队之间最大的区别。

在命令 – 控制式项目中，项目经理是负责人，也是进度表和计划的维护人。他代表的是利益干系人，负责收集需求、分解任务、从团队采集任务预估、分配任务并构建进度表。这就是 Roger 在 Lolleaderz.com 项目中采用的方法，他从 Avi 那里获得需求，从团队获得预估，然后给每个人分配任务。

命令 – 控制式项目中的成员很自然地会有一种 CYA 态度。如果别人的计划引出了问题，他们会很快撇清自己的干系。如果有人对进度和计划有绝对权力，那么团队中其他人和产

品所有者会欣然把决策都甩给这个人。

Scrum 项目中没有单独负责计划的人。这么做是有原因的。如果团队中负责计划和负责执行的人不同，那么一旦遇到麻烦，成员很容易互相指责。项目的绝大多数问题都和计划有关，所以这种麻烦总会出现在团队中。因此，Scrum 主管并不负责计划。他可以帮助团队制订计划。更重要的是，他会指导团队使用 Scrum 及其相关实践，让所有人觉得计划是大家一起制订的。Scrum 的实践和价值观帮助大家树立了主人翁意识。

4.3.2 产品所有者帮助团队了解软件的价值

想象一下，在 CEO 和 Scrum 主管之间有这样一段这样的关于项目的对话。CEO 问，在下一年内投入 200 万美元开发软件，他能得到什么。Scrum 主管告诉 CEO 说，他还不确定，他们每个月会汇报一下进度，在项目结束的时候交付至少价值 200 万美元的软件。没有哪个精神正常的 CEO 会批准这个项目。为什么呢？

CEO 绝对不可能批准这个项目，原因在于这个项目没有任何真正的*承诺*。这里的承诺指的是有人保证能完成某些事情，通常还要在特定的时间内完成。真正的承诺意味着责任和压力。如果发生状况，难以遵守承诺，许诺的人必须找到解决方案。

我们大部分人都有过这样的会议经历：会上有团队成员表示截止日期在制订时没有获得他们的认可，所以失败或超时并不是他们的问题。发生这种事情的原因在于：很多没有经验的项目经理会错误地认为，一旦把任务写到计划中了，整个团队就会自动对这个计划负责任。

计划并不会让我们承诺。我们自己的承诺才是承诺：计划只不过是一个方便记录这些要承诺的事情的地方。只有人才能做到承诺，项目计划只是负责记录。当有人指着项目计划说已经作出了承诺的时候，这个人并不只是在说一张纸而已。人们关心的是写在这张纸上面的承诺。

在 Scrum 团队中，产品所有者就是对公司作出承诺的那个人。这个人必须站出来，保证在项目结束的时能交付某个具体的东西。产品所有者负责的是项目应该满足的真正业务目标。产品所有者越能有效地让团队理解这些目标，对团队越有责任感，那么项目进展就会越好。如果项目不可避免地陷入了一些困境，例如遇到了技术困难、业务上出现了变动、或是有人离开了团队，产品所有者必须找到某种方式让团队理解当前最新的目标，唤起大家的责任感。产品所有者每天根据业务的变化做决策，每天都与团队碰头，确保大家都能理解积压工作表和项目目标正在发生的变化。

产品所有者不会坐等冲刺结束。产品所有者要对积压工作表负责并制订优先级、让团队了解业务情况、帮助团队理解积压工作表中哪些情景和条目最重要、最有价值，还要让大家理解一项积压工作表怎么就算完成了。在冲刺规划的过程中，整个团队共同从产品积压工作表中选择冲刺积压工作表，选择的依据是价值和必要工作量。产品所有者要在整个过程中指导大家，而且还不限于此。

产品所有者是一个很活跃的角色，要在项目中负责很多日常工作。像所有优秀的敏捷团队一样，Scrum 团队一定要有面对面的沟通，这样他们才可以准确地理解他们开发的东西。冲刺开始的规划阶段让大家为冲刺了解足够的信息，但是这还不够让产品所有者与整个团

队沟通清楚所有的细节。因此，在冲刺的过程中，产品所有者每天都会与所有团队成员一起工作，回答大量的细节问题，向团队成员就软件以及用户体验回答具体问题，还会作出很多关于产品工作方式的小决策。

产品所有者有作出这些决定的权力。（如果没有的话，他就不应该做这项工作！ Scrum 依赖产品所有者对业务决策的能力，包括验收最终结果的能力。）但是他并没有掌握所有的信息，因为很多用户和利益干系人也有有价值的信息和意见。因此，产品所有者还要花很多时间与这些人沟通，以获得开发人员所需要的答案。他还需要借助这个机会掌握一切最新的变更，并负责让产品积压工作表反映公司需求的最新变化。如果产品积压工作表的相对价值发生了变化，他要保证优先级得到调整，从而为团队下一阶段的冲刺规划做好准备。

4.3.3 每个人都对项目负责

Scrum 团队喜欢通过猪和鸡的寓言来帮助大家理解承诺是如何起作用的。

> 一头猪和一只鸡在路上走着。
>
> 鸡说："你好啊猪，我想我们可以开一家餐馆！"
>
> 猪说："哼，呃，也行啊，这家餐馆叫什么呢？"
>
> 鸡回应道："那就叫'火腿鸡蛋料理'怎么样？"
>
> 猪想了一会儿："我想还是算了吧。我得两肋插刀，而你管点蛋用就行了！"[4]

在 Scrum 项目中，谁是鸡，谁是猪呢？ Scrum 项目与低效的瀑布式项目相比，有什么不同呢？ Scrum 项目能不能成功，最终要看团队成员、项目经理和产品所有者如何工作。[5]

Scrum 团队经常会用这个故事讨论职责。鸡通常用来指项目中指派任务的个人，猪则是与项目生死与共的人。回想一下你自己的项目：你是不是总是把项目的成败看作自己的成败呢？

实际情况是，大部分职场中人都把自己定位为鸡。他们希望能对项目有贡献，但是他们认为把自己的工作（考评、升职、未来的职业目标、继续就业等）完全寄托给项目是一件很冒险的事情。另外，还有一点很容易忽视：人们为了钱而工作。为什么人要工作？真正激励他们的因素是什么？当前项目的成功并不总能解释一切。

在一个高效的敏捷团队中，所有的角色都是猪。每个人都真心感觉项目的成功就是自己的成功。但是缺乏经验的 Scrum 团队也很容易出现鸡的心态。

注 4：引用自故事 The Chicken and the Pig 的维基百科页面（网址为 http://en.wikipedia.org/wiki/The_Chicken_and_the_Pig，本文加入维基百科的日期为 2014 年 7 月 26 日）。

注 5：虽然听上去有点傻，但是 Scrum 团队真的谈论这个故事。事实上，一些比较老的 Scrum 指南中真的有一节专门讲猪和鸡的故事！



图 4-2：在猪和鸡的故事中，猪要两肋插刀，而鸡只是管了点蛋用

你作为一名开发人员，为项目选择一门技术时有几次是因为自己想学这门技术？作为一名项目经理，你是不是因为想提升市场竞争力而选择敏捷项目？也许吧。我们多少都做过这样的事情。每个人都可以自我激励，意识到这一点非常重要。但是 Scrum 对你有这样一点要求：冲刺的时候，让项目成功这件事情比你的其他任何职业目标都要重要[6]。换句话说，如果你在 Scrum 团队中工作，请把自己当成猪。

如果所有团队成员都是猪，那么说明大家都是两肋插刀的状态。这也意味着所有人都会为了项目的需要做任何工作。

下面举一个例子，帮助你理解这一点。假设你是一家公司的 CEO，而你全心全意投入项目。假设现在项目团队开始有一些倦意，很明显他们需要咖啡。如果所有人都在忙一些无

注 6：澄清一点：对于两肋插刀的人，项目成败比职业生活中其他任何事情都要挂心。当然，个人生活也有很多事情需要操心，例如家庭。如果事实不是如此，那么这个团队的思维方式可能存在问题，需要进行适度干预。

法脱身的事情，那么你会去给大家准备咖啡，这就是猪的状态。即使你是 CEO，你也觉得给大家准备咖啡是对你时间最好的利用，因为这正是团队现在需要的东西。

从另一个角度看，每一个软件项目都需要有鸡的存在。例如，每一位用户都是一只潜在的鸡。有多少次你因为浏览器或文字处理器或电子邮件客户端的一个特性而感到不满？你有没有在网站上发表过反馈的帖子或给支持团队发送过电子邮件？这是让你成为鸡的一种方式。如果团队听取了你的意见并修复了问题，你就帮助产品增加了价值。鸡在项目中参与的越多，给项目增加的价值就越多。

如果在 Scrum 项目里承担鸡的角色，那么你的意见会具有价值。你关心的是项目的产出，团队希望能听到你的声音。但是你的工作并没有绑在项目上：你还有其他的目标（例如出售产品、支持产品或运营这家公司）。这些目标都很重要，但是并不等同于项目需要产出的具体工作。

这就是为什么 Scrum 团队（特别是产品所有者）注重与鸡培养关系。最有效的做法就是定期或按照可预期的进度给用户发布可工作的软件。这样可以让鸡持续参与，并且帮助他们看到自己对项目的影响。

1. 怎样让产品所有者、Scrum主管和团队成员成为更好的猪

如果团队成员只是完成项目经理指派的任务时，那么当计划出现问题的时候，他不会觉得那是他的问题。他只是像鸡一样工作。他并没有觉得自己全心投入了这个项目，更重要的是，他没有感到要对团队其他成员负责。换句话说，如果这名团队成员真心觉得有责任把计划执行好，还有责任尽可能地开发出对公司和团队最有价值的软件，那么他就是像猪一样工作。

遗憾的是，很多公司都希望团队成员表现得更像鸡，而不是猪。事实上，程序员在试图参与项目计划制订时，常常会被排除在外，因为计划和决策是经理的特权，而程序员这样的普罗大众是没有机会参与的。（“你以为你是谁？你算哪门子经理啊？快回到你的小工位去吧，卑微的程序员！”）如果一家公司的文化包含这样的价值观，那么这家公司的团队就很难真正用上 Scrum。

Scrum 主管很容易一不小心就鼓励整个团队成员都成为鸡，让自己成为计划的所有者（或是负责人）。对于正在学习想要成为一名 Scrum 主管的传统项目经理来说，发号施令的老习惯很难改变。很多项目经理在职业生涯中有很多负责计划实施的成功案例，但这些成功并不会对成为 Scrum 主管有任何帮助。上层的管理很容易让一个人负责全部事情（有些人喜欢把这种人称为*独掌生杀大权的人*）。做这个人一定非常有成就感，因为这个人能感到自己的用处和重要性，他似乎以秩序平息了秩序。

计划很有必要，但是把计划交付到团队成员的手中就没有必要了——如果一位 Scrum 主管自己将一个冲刺的工作分解，通过团队得到工作量预估，把工作分派给团队成员，再检查团队成员的状态，那么这位 Scrum 主管在扮演计划负责人的角色。同时他也在鼓励团队成员成为鸡而不是猪。

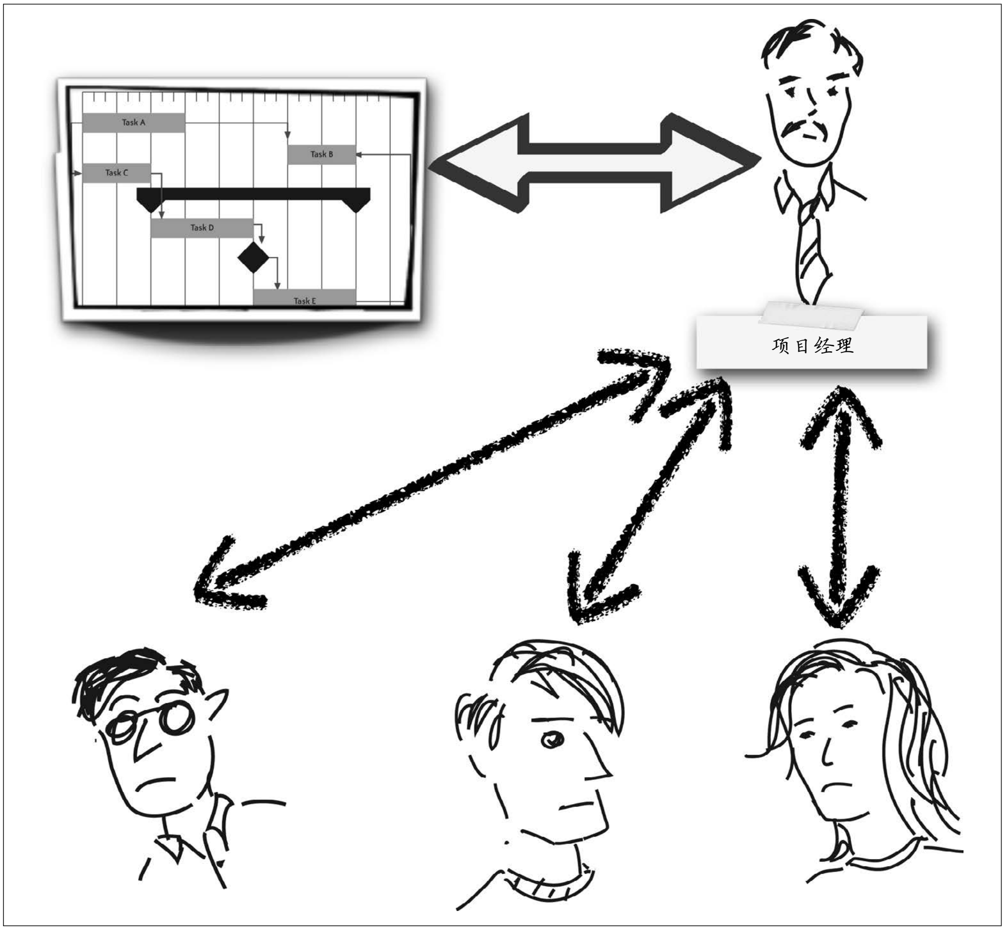


图 4-3：命令 – 控制式项目经理实际上把自己当作了计划负责人，他鼓励团队成员成为鸡

Scrum 主管在引入 Scrum 时最大的一个障碍就是要纠正团队中像鸡一样工作的氛围。团队往往喜欢这个命令 – 控制式项目经理的角色，因为这样他们就不用过问整个项目了。但是考虑整个项目是优质高效完成项目的关键。如果开发团队在编码的时候短视，那么构建出来的就是设计糟糕且脆弱的系统。如果团队偷工减料，应付差事，那么他们构建出来的东西可能当时可以正常工作，但是整个系统都脆弱而难以维护。

Scrum 主管应该把工作量的预估当作待寻找的事实，而不是对团队的要求。Scrum 主管通过这种方式鼓励团队成员有猪的精神。对一项任务的估计应当遵循实际情况，因为团队的确要花一定的时间完成工作。最好的预估要符合实际，不能随便画大饼，随便猜一个结果取悦老板或是尽早结束会议。

一个好的计划就像是一段还没有书写的历史。团队在冲刺结束团队时进行回顾，应该可以完全准确地说出他们都完成了哪些任务，每一项任务都花了多少时间。冲刺会发生在计划之后，在这一轮冲刺中发生的任何事情都是已经确定的信息，即记录在案的事实。如果他

们任务完成得好，那么冲刺初期指定的计划会与冲刺结束时的事实非常接近。计划与最终发生的事实越接近，那么这个计划就越准确。

对于很多团队来说，这是一种思维方式的变化。将计划视为一组乐观目标的团队往往苦苦挣扎却无法完成任务。毕竟，一个过于乐观的团队成员或经理就足以毁了大家的计划（以及接下来的一个月），纵然他认为自己当时是在帮忙。不过，如果团队在制订计划的时候，尽最大努力写下接下来 30 天真正要做的事情，那就不必打肿脸充胖子了。这样也极大地降低了一味赶进度，开发出脆弱代码的可能性。

项目经理需要用全新的思维对待计划。传统的项目经理往往把计划当作激励，强迫团队在截止时间内完成任务。他们经常把计划看作让团队找到事情做的方式，仿佛如果没有计划，团队就会整天坐在那里无所事事。在冲刺开始的时候要求团队作出激进预估并"批准"计划的项目经理事后很容易利用计划压榨团队。"制订计划并按计划工作"的工作方式就这样在项目经理与团队之间产生了不和。

在一个高效的 Scrum 团队中，Scrum 主管不会要求大家预估自己的工作量，让每个人对自己的估计负责，而是会与整个团队一起来做每一项预估。他不只是把任务分配出去。Scrum 主管和团队一起尽力估计未来可能发生的事情，并且始终共同努力让计划尽可能准确。这样团队才有足够的时间让以最好的状态完成工作，从而最终开发出最有价值的软件。

如果在冲刺计划阶段参与工作分配的过程，而不是等着被分配任务，那么团队成员也会有更多的参与感。事实上，如果团队成员能真正投入进去，冲刺初期甚至不需要分配任务。高效的 Scrum 团队在冲刺的过程中可以根据每个人的空闲时间和技能决定谁来完成哪些任务。这是理解自组织团队工作方式的要点之一。

这就意味着，在高效 Scrum 团队中，团队成员并不满足于完成自己的任务。每一个人都发自内心地想要给用户和利益干系人交付出最有价值的软件。当团队中每一位成员都有这种心态的时候，我们才能认为团队作出了*集体承诺*。在这种情况下，并非每个人都领到了零碎的任务，而是整个团队投入了开发交付积压工作表中最有价值的过程。在开发过程中发现了与项目有关的新情况，团队会有足够的自由调整完成任务的方式。例如，如果在项目进展到一半的时候，Roger 和开发人员意识到有些功能的开发方式需要改变，那么 Avi 完全可以信任他们，因为他相信他们能够交付冲刺积压工作表中的所有项目。如果最后发现他们搞错了（真的搞错了！），计划完不成，他们会与 Avi 沟通他关心的积压工作表，而不需要对每一个小细节作出解释。

这是 Scrum 起效的根本原因之一，这也决定了高产 Scrum 团队与那些"聊胜于无"的团队之间的本质区别。

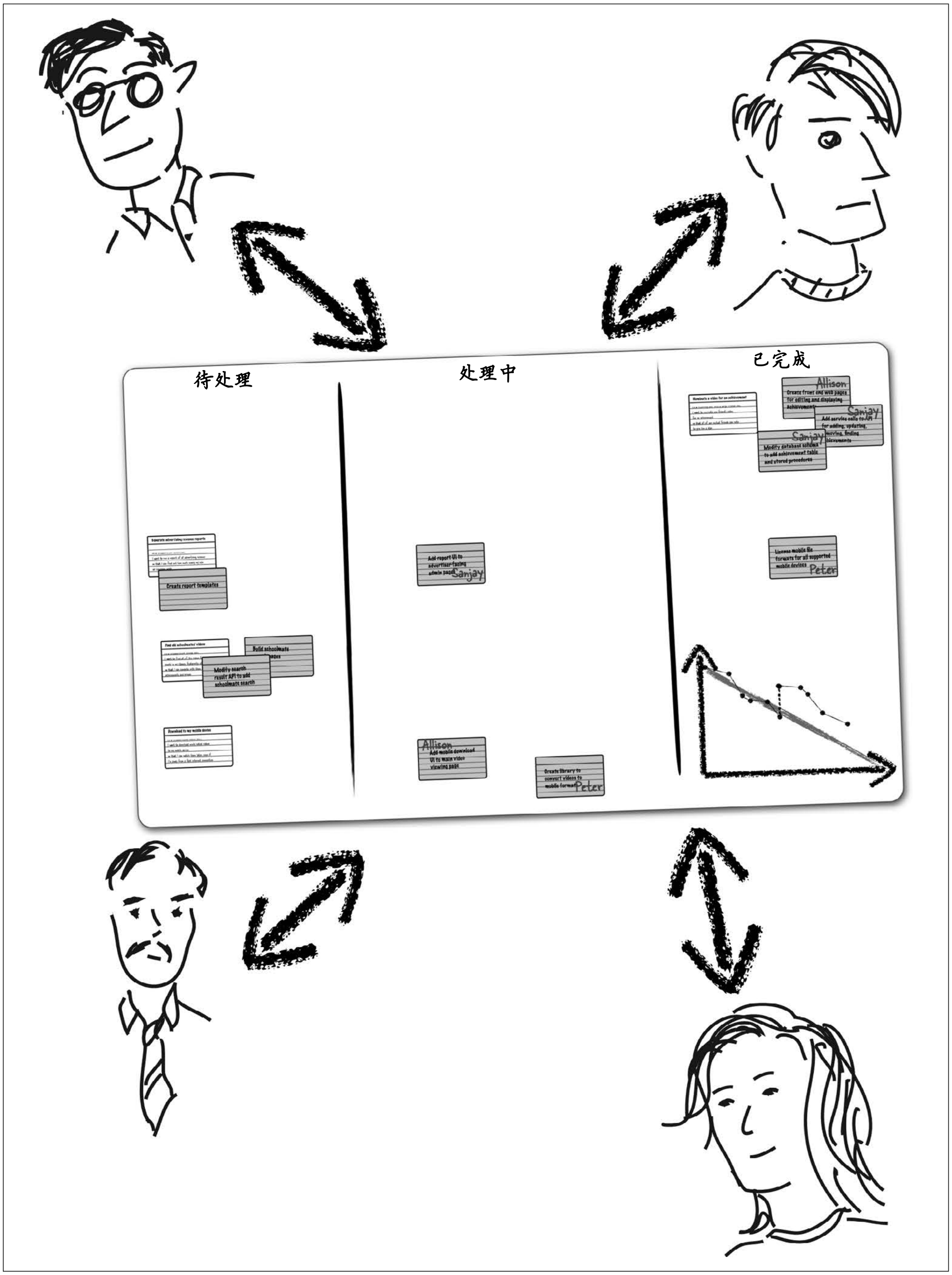


图 4-4：自组织的团队把预估和计划当作待寻找的事实，而不是借此逼迫整个团队

2. 没有鸡的空间

Scrum 团队并没有给鸡留出空间。产品所有者也属于团队的一部分，因此他们也需要当猪。对于产品所有者来说，这一点并不好接受，特别是在他们认为分到 Scrum 团队很倒霉的时候。大部分利益干系人很自然地想当鸡，因为距离越远越舒适。（有时候他们又想多管闲事，因为这样可以给他们带来对团队施加影响力的权威感。这是产品所有者需要解决的问题。）

Scrum 主管和团队有一些办法可以让产品所有者拥有真正的投入感。最重要的方式是用心倾听产品所有者的观点和意见，并且意识到产品所有者会给项目带来团队真正需要的专业知识。

很多程序员都认为在项目中只有编程才是最重要的事情，其他所有的事情都应该放在技术细节之后。更糟糕的是，很多公司还鼓励这种误解，这些公司的组织架构是围绕着技术人员展开的。项目经理和产品所有者从技术团队分开的公司组织架构会让程序员把其他人当作"局外人"，并且不尊重他们的贡献。

产品所有者并不一定了解技术细节，但没有问题。在 Scrum 团队中，每个人都有自己特有的技能和知识，负责最适合自己的任务。产品所有者让大家对项目的目标有真正深入的理解。Scrum 主管和团队与产品所有者沟通越多，询问项目目标越多，越理解产品所有者的观点，那么产品所有者对项目和团队的投入也越多。

4.3.4 Scrum有一组自己的价值观

每一家公司都有自己的文化，其中包含了一些特有的价值观。例如，有的公司看重职责分离，即每个人都有自己特定的角色。如果无法轻易影响或控制一件事，那么这个人就不需要对这件事负责。还有一些公司看重的透明，信息可以自由分享，基层员工也能影响管理层的决策。不能说哪一种一定是正确的。每一家公司都有着自己的历程和文化，这反映了公司的管理和决策方式。

每一套方法都有内在价值观。我们在第 3 章了解到，具体的敏捷原则往往都绑定了一些独立的实践（或是说由这些实践实现），而这些实践则是团队在项目中运用每一项原则的有效方法。我们已经知道，如果一家公司中的团队成员让经理享有绝对的决策权，那么团队就会有一种置身事外的感觉。价值观和原则也是一样的：与公司意识形态冲突的价值观会遇到障碍。

但是如果一家公司的文化与敏捷价值观和原则匹配，那么这家公司中的敏捷团队就会比命令 – 控制式团队更容易获得成功。（一些敏捷团队之所以报告了"令人惊讶的结果"，这也是原因之一。）

你可能会很惊讶地发现敏捷价值观和原则与你公司的文化非常匹配。

在公司中引入敏捷的首要步骤是对敏捷价值观进行讨论，看看这些价值观会对公司的文化带来什么样的影响。如果你发现在公司中引入敏捷的过程遇到了麻烦，那么就要找到敏捷价值观与公司文化的差异所在，这样有助于平缓过渡。（至少可以在感觉上好一点，因为你能理解哪里出了问题。）

自组织团队和命令 – 控制式团队工作方式上的差异来源于价值观的不同。在《Scrum 敏捷项目管理》一书中，Ken Schwaber 讨论了 5 种 Scrum 价值观：勇气（courage）、承诺（commitment）、尊重（respect）、专注（focus）和开放（openness）。学习这些价值观并理解自组织团队是可以在项目中切实开展的工作。

- 每一位成员都承诺实现项目的目标

  如果团队有权利作出实现这些目标的决策，并且所有项目成员都对于项目的规划和执行有发言权，那么团队就可以达成这个层次的承诺。第 3 章中的电子书阅读器团队一开始的时候就得到了一个要开发一个网络书店的需求。为了让产品成功，他们不得不忽略一开始的需求，这样才能交付一个更有价值的项目。只有在团队、Scrum 主管以及产品所有者能够作出决策的时候才能真正做到这一点。他们不需要层层官僚审批就可以作出这样的决策。

- 团队成员互相尊重

  团队成员只有互相尊重才会信任同事能把手里的工作做好。程序员和其他专业人员之间并不容易做到这一点。很多程序员，特别是技能水平很高的程序员，往往纯粹根据技术能力建立尊重。这是高效采用 Scrum 的一个障碍。如果程序员不尊重产品所有者，讨论到项目目标的时候他就听不进产品所有者的意见。

  好的 Scrum 主管能够找到让团队成员相互尊重的方法。例如，他会让程序员知道产品所有者对用户思考方式以及公司的需求有着深入的理解。当程序员明白这些知识对于项目的成功十分重要的时候，他们就知道产品所有者的价值并尊重他们的意见。

- 所有人都专注于工作

  当前冲刺是 Scrum 团队成员一个时期内唯一的工作。为了完成冲刺积压工作表中的工作，并且处理在冲刺期间发生的任何需求变更，个人可以随意采用自己需要的工作方法。当团队中所有成员都专注于冲刺目标并且有足够的自由完成满足这些目标所需要的工作，那么整个团队就能够发生自组织，在需求变化的时候很轻松地调整方向。

  从另一方面看，目标分散的团队效率更低。在现代职场（特别是在程序员中）存在一个误解，那就是多任务工作效率最高，因为当一个项目受阻的时候你可以转战另一个项目。然而实际情况并不是这样！在项目之间互相切换，甚至在同一个项目中的不同任务之间切换，都会增加意想不到的延迟和代价，因为背景切换需要消耗大量的认知。把手头的工作放下，然后拾起另一个项目中断的工作，这个过程消耗的脑力资源是超出想象的。切换任务的时候，需要回想上一次离开的时候要解决的是什么问题。要求一个团队切换到另一个项目中的某项工作，不仅要考虑新任务所需的时间，还要考虑任务来回切换所耗的时间，而这个时间几乎与做这项任务所需的时间相同。

  不信？好好想一想，假设你得到了两项为期一周的任务。假设有某种神奇的定律可以保证多任务不会产生任何额外成本。你可以在这些任务之间无缝切换，不会有一丁点开销或延迟，因此这两项任务正好需要两周的时间。即使是在这些完美（却不可能）的情况下，执行多任务也没有什么意义。如果你不执行多任务，你可以在第一周结束的时候完成第一项工作，并且在第二周结束的时候完成第二项工作。然而，如果你采用多任务执行，那么你肯定会在第一周里面花一些时间完成第二项任务，因此第一项任务肯定要到

第二周的某个时刻才能完成。因此，即使人类擅长处理多任务，也没有必要真的这么做，何况我们并不擅长这个。

会让团队成员分心的不只有多任务。他们经常会被要求参加各种无用的会议和各种不相关的组织，做一些与项目无关的事情，还要为其他项目提供支持。好的 Scrum 团队有权忽略这些让人分心的事情，而不用担心影响自己的事业或晋升。[7]（当前项目相关的支持工作可以加到冲刺积压工作表中，先完成别的任务，让项目整体按时交付。）

- *团队看重开放*

  在 Scrum 团队工作的时候，团队中的其他所有人都应该了解你当前正在做的工作，以及你正在做的工作是如何朝着当前目标前进的。基于这个原因，基本的 Scrum 工作模式中的实践都会鼓励团队成员间的开放。例如，通过任务板，大家可以看到其他所有人要做的工作，以及还没有完成的工作量。通过燃尽图，大家可以自己度量冲刺达成目标的速度。高效的每日 Scrum 是纯粹的开放性实践，每个人都会分享自己的任务、遇到的困难以及当前的进度，让整个团队都了解这些事项。所有这些做法都有助于团队培养互相支持和鼓励的氛围。

  在 Scrum 团队中营造开放的文化听上去很棒也很正能量，而且事实上就是如此。但这有可能是 Scrum 团队中最难达成的一件事情，因为开放是 Scrum 与公司既有文化之间常见的冲突点。

  很多公司都会阻碍透明的文化，把 Scrum 的开放精神替换为阻隔信息的层级制度。建立这种文化，经理可以从多个方面获得好处。缺乏透明的组织更容易要求团队去达成一个不切实际的目标（“我不管你们怎么做，一定要做好！”），并迫使团队加班。当团队实在是无法达成目标的时候，这也给经理带来了推卸责任的借口（“这都是他们搞砸了，不是我的错！”）。

  正因如此，开放和自组织往往会被人称为 Scrum 实践中不可触碰的雷区。这既是确保正确实施 Scrum 的核心概念，也要求公司对待团队的方式异于传统。不透明的经理长期暴露在开发相关的细节中会失去 CYA 态度的保护。在很多新手 Scrum 团队中，当不透明的经理开始了解具体开发细节时，他们发现他们采用 Scrum 的努力从上层开始被破坏了。

  开放性对于那些油头粉面、华而不实的经理来说是一种威胁。不过从现实情况看，优秀的团队在引入 Scrum 的过程中也会遇到这种困难。假设有某个开发人员被看作是某一部分代码的专家，或者某个项目经理是计划的“守护者”，或者某个产品所有者是大量用户的唯一联络人并且负责作出软件相关的重要决策，想象一下从这些人的角度看开放意味着什么。这些人都有权把这些工作看作是自己对项目的贡献。把这些工作开放给整个团队会是一件困难的事情，这会鼓励团队其他成员分享参与这些工作的权限，甚至在未经允许的情况下变革。

注 7：这条建议看上去是不是不现实？有没有触碰某根神经？如果团队无权忽略这些分心的事情，那么可能这就不应该被认为是分心的事情。不能忽略的分心事就是需求，而根本不能算是分散注意力的事。如果你工作的团队把忽略“分心事”当作严重的项目问题，那么团队的思维方式可能对采用 Scrum 来说是一项阻碍。这需要你在团队中多付出一些。

这是团队成员个人反感开放性的一个非常自然的理由。但是如果他们跨过这个坎，开始与整个团队一起工作，那么所有人都能获益，因为这是互相信任并快速交付更有价值软件的唯一方式。

- 团队成员有勇气全力支持项目

  选择了开放性，摒弃不透明，你就能让团队更强大，而不是消耗整个团队来让你自己更强大。做到这一点需要勇气，但是一旦豁出去，你最终得到的会是更好的产品和更好的工作环境。

  Scrum 团队有勇气坚守对项目有利的价值观和原则。要应对与 Scrum 以及敏捷价值观冲突的公司阻力还是需要勇气的。为此，每一位团队成员（尤其是 Scrum 主管）都需要保持警觉。每个人都要相信，交付有价值的软件可以帮助自己克服针对这些价值观的阻力。这也是需要勇气的，特别是坐下来与老板一起做回顾工作的时候。你要勇气对自己说："帮助这个团队产出更有价值的软件比吹嘘我的个人贡献吹嘘更重要。"

  那么怎样给团队带来勇气呢？怎样为团队成员建立自信，相信 Scrum 不仅可以帮助他们开发更有价值的软件，还能让公司看到他们采用的新方法的价值？

要点回顾

- Scrum 项目的基本模式包含了 Scrum 的角色和实践：Scrum 主管、产品所有者、团队、冲刺、产品积压工作表和冲刺积压工作表、每日 Scrum 例会、审查和回顾会议。
- 为了真正做到 Scrum，团队成员不能仅满足于相关实践，还要真正理解**自组织和集体承诺**。
- 团队通过猪和鸡的故事来理解如何承诺交付有价值的软件，由此知晓如何对整个团队产出的每一件东西都有主人翁意识。
- 团队需要真正理解并吸收以下 Scrum 价值观：**承诺**、**尊重**、**专注**、**开放**和**勇气**。做到这一点的才是真正的 Scrum 团队。

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试引入敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

4.4 第2幕：状态更新只是社交网络的玩法[8]

回到 Lolleaderz.com 团队，Roger 和 Avi 需要帮助，他们也意识到了这一点。他们还知道

注 8：本标题的意思是说，"状态更新"这种事情在社交网络中玩玩就好了，不要用在项目管理中。——译者注

在 Hover Puppy 有另外一个团队在 Scrum 方面经验丰富。Roger 找那个团队的人聊天，想知道他们有什么秘诀，谁知聊到最后比一开始还要困惑。他们做的事情貌似与 Lolleaderz.com 团队做的事情完全一样：他们也有冲刺、每日 Scrum 例会、回顾会议、积压工作表、产品所有者以及 Scrum 主管。看上去两个团队做的事情都是完全一样的，但是一个团队收获了很好的结果，而另一个团队只能慢慢挣扎。Roger 和 Avi 同另一个团队的 Scrum 主管 Eric 聊了一次。他在 Scrum 方面有很多成功的经验，也很乐意帮助 Avi 和 Roger 找到问题所在。

Eric 问的第一个问题是："你们有没有教练？" Roger 不是很明白 Eric 的意思。"不知道吗？就是导师啊——能帮你正确完成 Scrum 的人。没有教练的话，我们的团队不可能像现在这样好。"这是 Roger 第一次真正认识到 Avi 的销售技能有多么棒，因为在讨论结束的时候，他已经说服 Eric 签约成为 Lolleaderz.com 团队的 Scrum 教练。

接下来的周一，Roger 和 Avi 想在每日 Scrum 例会上隆重向大家介绍 Eric。Eric 让大家顺其自然，他会观察团队的工作，然后尝试给出一些小的建议。这是件好事，因为在介绍会上团队只有一半人在，他们都知道主开发人员肯定最先发言，而他的汇报通常最长。因此，其他团队成员一般都等他讲到一半才来。

在会议剩下的时间里，成员轮流向 Roger 汇报工作，讲明被安排的工作的进度，并向 Roger 请示接下来的任务。在汇报过程中，有一个人指出他们仍然在等待系统管理员修复某一台 Web 服务器上的配置错误，问 Roger 打算怎么把这个错误搞定。Roger 把这项工作加到自己要为团队清除的障碍列表中。Eric 在一旁观察了完整的会议，没有说一句话。

第二天，Eric 旁听了另一个每日 Scrum 例会，这个例会与前一个例会流程完全一致。他注意到有一位团队成员报告说他完成了 95% 的任务，而上一次会议这个人也是这么说的。会后，他找 Roger 问起此事。"是的，看上去确实会出问题。不用担心，一切都在掌控中。我已经更新了计划表。他总是会拖延任务，所以我留了足够的余量。如果拖延得太多了，我会保证把这件事情报告给 Avi 和 CEO。"

当天晚些时候，Eric 和 Roger 以及 Avi 召开了会议。他首先讲解了一下他认为的最大问题。"Roger，你只是把每日 Scrum 例会当作管理计划表的一种方式。如果那位团队成员一直拖延怎么办？你只是更新你的计划表。好，你的工作做完了，对吧？但是仅仅更新一下某一个甘特图并不能让项目拖延得到缓解。"

Roger 听到这种说法并不高兴。Avi 也感觉不舒服，因为他在用这份计划表给项目其他利益干系人汇报最新情况。Eric 继续说着，告诉 Roger 他只是在利用每日 Scrum 例会获得团队的最新情况。Roger 听了更不高兴了。"我当然是用例会来获得最新进展了！这不就是这个例会的作用吗？" Roger 开始怀疑当初该不该让 Eric 来做教练了。

你能不能指出为什么 Eric 会认为 Roger 和 Avi 把每日 Scrum 例会当作团队进展汇报是有问题的？根据这些汇报的最新情况更新计划表有什么问题？如果每日 Scrum 例会的目的不是这些，那么 Scrum 团队为什么要开这个例会？

4.5 整个团队参与每日Scrum例会

每日 Scrum 例会是 Scrum 团队可以利用的最有效的工具之一。这是因为每日 Scrum 例会帮助团队完成了两件重要的事情：首先是*检查*（inspection）团队正在做的事情，帮助调整工作以交付最大的价值；其次是让团队有机会在*最后责任时刻*（last responsible moment）作出决策，使得团队具有足够的灵活性，可以让正确的人在正确的时间完成正确的工作。当团队中每一位成员都利用每日 Scrum 例会讨论下一轮迭代要开发的必要功能时，做计划所需要的工作就会得到限制，那么整个团队就会开始意识到每日 Scrum 例会是一种有价值的工具并且开始高效地使用这种工具。

4.5.1 反馈和“可见–检查–调整”周期

很多刚接触敏捷的开发人员还习惯于整个世界都围着编程转。他们参与并投入开发工作。这是开发人员的舒适区，因为他们可以完全忘我地解决技术问题。

但是每一位程序员都有过这样的体验：花了很多时间和精力开发出一套解决方案，结果发现有重要问题遗漏，原因通常是从来没有人告诉他们这是个问题。

在传统的*大需求在先*（Big Requirements Up Front，BRUF）项目中，这是一件特别成问题的事情。考虑一下，在一个这样的项目中，一项需求在到达开发人员之前需要经历哪些步骤。典型 BRUF 瀑布式项目工作的规划流程如下。

- 项目经理需要考察要做的工作，通常采用的方式是研究某些业务需求文档或开发范围，并开发目标文档。
- 经理需要签字确认开发范围。
- 业务分析师需要对开发范围进行审核，然后与用户以及其他利益干系人交谈以理解他们的工作。
- 业务分析师得出用例以及功能需求等。
- 程序员根据这些需求预估工作量。
- 项目经理根据需求和预估的工作量建立计划表，并且与利益干系人和经理一起审核。

可以看出，在真正开始开发之前，还有一长串事情要做。这就不难理解为什么很少有需求能经过这样的“传话游戏”，完好如初地到达开发人员耳中。

这绝不是 BRUF 瀑布式团队特有的问题。全然面对面沟通的团队也会遇到误解和沟通不明确的问题。面对面沟通效率很高，但是精确性还是比书面沟通要差。很容易出现这样的现象：三人相谈甚欢，以为达成了一致，结果每个人对讨论的内容都有不同的印象。

有句老话说：“阳光是最好的消毒剂。”尽管从医学的角度看，这句话可能有问题，但这对项目团队来说是一个非常好的建议。让用户评判团队是否开发出了有价值的软件的最佳方式就是尽可能频繁地把可工作的软件交到用户手中。这就叫*可见性*（visibility）或*透明性*（transparency），这个道理也能用在沟通上。

每日 Scrum 例会是一个非常有效的可见性工具，因为这可以帮助解决这些与沟通相关的问题。每当发生这些问题的时候，每日 Scrum 例会就会派上用场。以上面那三个人为例。如

果这三个人都停下来，然后开始完成三个不同的任务，冲刺结束后进行集成时会怎么样？这些小小的误解会在项目进行过程中慢慢产生问题和裂隙，如果不及时发现，最后会产生堆积如山的问题。这里或那里的小误解可能会产生一两处的瑕疵，事后发现这些瑕疵的时候，肯定要想办法除去，因此需要一个快速补丁或较大的修改。如果长期项目出现这种问题，那么代码库的质量会随着时间的推移而逐渐恶化。

从另一方面看，想象一下如果这三个人每天都花 15 分钟时间坐在一起互相提出以下这些问题。

- 上次会议结束到现在，你都做了些什么？
- 从现在开始到下一次会议，你计划做什么？
- 有哪些阻挡你前进的困难？

如果成员每天都可以用同样的方式在团队内公开自己的工作，那么由于不可避免的沟通错误而导致的问题很多都可以扼杀在萌芽状态。如果团队中每一位成员都*检查*其他人正在做的工作，那么他们就可以共同得出结论，并且对于项目的目标和大家实现目标的方法达成一致。

在每日 Scrum 例会中，如果一个人讲述了自己手头的工作，那么他的同事也许会给出某些方面的改进建议。如果建议很不错，那么他接下来一天的工作就可以干得更好。同事也有可能发现他在做一项完全错误的任务，而这个问题极有可能是由误解而导致的，那么这种发现会让他改变接下来一天工作的计划。这些改变称为*调整*（adaptation）。通过可见 – 检查 – 调整的每日循环，团队可以不断地通过项目反馈来改善自己开发软件的方式。这是 Scrum 的一项最重要的特性。Scrum 团队根据项目经验以及项目中实际发生的事实来做决策。[9]

通过这种类型的反馈机制，Scrum 团队可以摆脱与实际工作脱节的项目经理，可以降低“传话游戏”式的沟通带来的损失，还能在提升质量的同时节约时间。这种循环是一种重要的*反馈闭环*（feedback loop），团队通过这种闭环让项目保持在正轨上，还可以保证所有人的想法都一致。

对于程序员来说，检查其他所有团队成员的工作并不令人感到舒适。然而，即使是最内向的团队成员通常也能习惯这种指责，还有很多人最终会很期待每日 Scrum 例会。这是因为对于 Scrum 团队来说，每日 Scrum 例会是一种让大家沟通并共同理解工作的最高效方式。

4.5.2　最后责任时刻

差不多到了这个时候，聪明且多疑的命令—控制式项目经理可能会有这种想法：“好，我懂了。在 Scrum 项目中，团队不会一开始就声称自己了解一切。沟通和一致的理解对团队来说很重要。但是事情还是要去做，任务要分配到人。怎样才能真正把工作完成？到底要通过什么样的机制把实际的编程任务、数据库任务、测试任务以及其他所有任务放到程序员的待处理事项列表中？”

注 9：Scrum 理论的学生把这个过程称为经验式的过程控制，有关经验主义的更多信息请参阅 *Scrum Guide*（https://www.scrum.org/）。

为了指导团队更好地进行每日 Scrum 例会，培训师和敏捷教练经常采用的一个技巧是“让他们失败”[10]。当一个习惯于命令－控制式管理的团队开始进行每日 Scrum 例会的时候，成员往往会指望项目经理或 Scrum 主管对会议进行引导。这样一来，项目经理会询问每一位团队成员的最新状态，然后给他们分配下一项任务。对于习惯接受分配、完成任务的团队来说，这感觉非常自然。尝试帮助团队有效执行每日 Scrum 例会的敏捷教练会建议 Scrum 主管在这一次会议中不要发表意见。这样做往往会导致一两分钟尴尬难捱的寂静。最后终于还是有人开口，介绍自己自上一次会议以来所做的工作。如果这个团队幸运的话，这位团队成员接下来要问的问题就是“我下一步的任务是什么？”

这就是真正区分 Scrum 主管和普通项目经理的时刻了。项目经理会继续给团队成员一项预先分配好的任务。而 Scrum 主管则意识到这是一个让团队领悟敏捷的机会，应该让大家理解任务到底应当如何分配。他会提出这类问题：“你觉得下一步应该做什么事情？”当然，根据团队的实际情况，他也可以继续保持沉默。

要点在于，任务分配取决于团队成员。*每一位成员在完成了当前的任务之后自己给自己分配下一项任务。*

任务分配这件事情应该在每日 Scrum 例会上完成，因为团队其他成员有机会给出自己的意见，帮助整个项目运转顺利。例如，如果有一名开发人员自告奋勇揽下一项复杂数据库优化的任务，那么 DBA 可能会站出来建议这名开发人员先不要做这项任务，并承诺他接下来会负责这项任务。

且慢，难道我们不能通过事先计划的方式来避免瓶颈吗？

开始分配项目时，命令－控制式项目经理会假设每一项任务都必须由某一个特定的团队成员来完成，分配的依据通常是这个人的专业知识或技能（例如 DBA 具有数据库优化的技能）。这么看起来事先做好计划也是有道理的：某个人负责的任务构成了进度表中的一个瓶颈，那么为了赶上截止时间，团队必须围绕着这个瓶颈做计划，考虑如何避开它。

讽刺的是，这种做法最后成为了项目管理问题的常见病根。非常复杂的任务比简单的任务更难预估。而依赖于某个特定人或资源的任务相比可以由多位团队成员完成的任务风险更大。因此，像这种必须由某个特定的具有某种技能的团队成员完成的复杂任务（例如复杂的数据库优化任务）最有可能产生错误的预估，这不应该令人感到意外。更糟糕的是，项目经理通常非常依赖这位有技能的团队成员，不仅依赖他的工作，还依赖他的预估。

我们不可能事先预料到一切。在项目开始的时候，你需要作出一些决策。（用 Java 还是 C#？用 Windows 还是 Linux？用 Mac 还是 PC？）甚至还有一些任务绝对只能由某个特定的人来完成。但是一般情况下，Scrum 团队不会在项目开始的时候分配好任务，甚至不会在冲刺开始的时候完成任务分配。

事实上，Scrum 团队根本不会试图得出一个“最终”的任务序列。原因在于，对于大部分项目的任务（尤其是编程任务）来说，团队并不能真正提前知道需要花多长时间。而且，往往只有碰到相关依赖的时候，团队他们才发现有这些依赖。在项目进展的过程中，团队

注 10：Lyssa Adkins 在《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》一书中讨论了这项技巧和其他相关技巧。

经常会发现遗漏的任务。另外，一开始看起来很小的任务到最后可能被发现是一项重大任务，一开始看起来很重大的任务最后被发现微不足道，这些都是很常见的事情。当然，尽管团队经常会在冲刺过程中发现漏掉了任务，但这并不表示他们没有责任在冲刺规划的过程中尽可能制订出完整的任务清单。

因此，敏捷团队在冲刺开始的时候，并不会把要做的工作分解为任务，对任务排序，在没有开始做任何任务之前把每一项任务分配给团队成员，然后跟踪执行指定的计划。敏捷团队遵循一个简单的计划原则。他们在*最后责任时刻*作出所有的决策。

回到 Lolleaderz.com 项目的故事。Roger 和 Avi 在第 4 轮冲刺中遇到了问题，因为 DBA 在某一项需要特别技能的任务中耗费了更长的时间，而这些时间似乎本应该在一开始就正式计划好。这过度计划导致的一个常见的项目失败。项目经理假定一组任务由某个人完成，这个人通常是有某些特别技能的专家。这些任务往往是具有最大拖延风险的任务。当这些任务不可避免地拖延的时候，其他人也没有能力完成这些任务，因此这样的拖延会产生连锁反应，导致项目延期，更常见的情况是，大量加班导致专家的工作效果变差（甚至开始寻找新的工作！）。

Scrum 团队在最后责任时刻作出决策，因此处理这种事情的方式会有所不同。Scrum 团队不会在冲刺一开始的时候假定 DBA 会完成所有这些任务，而是把这些任务写在索引卡（或某些对应的电子产品）上，然后把索引卡放到任务板的“待处理”栏里。在每日 Scrum 例会上，所有人都可以看到这些索引卡，一般都会有人指着这些索引卡询问这些任务是否会在冲刺后期导致问题。

在开源团队中有一种说法：“基于足够的关注，所有 bug 都无所遁形”，这就是 Linus 定律的思想。计划也是如此，每日 Scrum 例会就是让大家关注计划的机会。如果整个团队像看代码一样认真地看项目安排，那么从进度表中筛查出弊病的可能性就更大了。在每日 Scrum 例会中，团队在项目进行过程中发现瓶颈的难度比命令 – 控制式项目经理在一开始就看出瓶颈的难度要低多了。如果能看到瓶颈，那么团队也能找到绕过瓶颈的方法。每个人都在审视项目，所以团队常常会发现有一些任务在冲刺的最后责任时刻比其他任务要简单得多。这就是“可见 – 检查 – 调整”周期的价值所在：让团队有机会找到潜在的问题，然后共同找到解决方案。

那么，如果要召开更有效的每日 Scrum 例会，Roger 和 Avi 该怎么做呢？他们不应该给团队分配任务，而是应该利用会议的时间让大家能在一起找出进度安排上的问题，并且让大家给自己分配任务。他们可能之前已经意识到——他们以成员的身份和团队一起工作和讨论——为了能赶在冲刺结束的时候完成任务，需要找到除 DBA 之外的人负责存储过程相关的工作。如果实在不行的话，他们至少知道了要求的事情多过了能做到的事情。这样一来，他们可以在冲刺早期给大家设定好合适的心理预期，提前知道在冲刺结束的时候能交付出什么样的可工作软件。

4.5.3 召开有效的每日Scrum例会

- *表现得像猪一样*

  在会议中，每一位团队成员都要对自己的队友负责。如果在前一次会议中作出的承诺没

有实现，那么你要负责解释具体的原因。想象一下自己在一个自组织的团队中，你真心对项目感到有责任感，那么为了做好自己的日常工作，你需要了解哪些情况？最重要的是知晓你手头正在做的工作。但是如果你的团队是真正自组织的，那么就不能指望有一个命令 – 控制式项目经理来帮你决定你要做什么。你需要通过一些其他的机制来获得接下来的任务。因此，你在每日 Scrum 例会上的第一项收获就是下一项任务。轮到你讲述下一次 Scrum 例会前的工作计划时，如果你已经完成了到目前为止分配给你的所有任务，那么你要做的就是看一下“待处理”列表中的任务，然后从中挑选出一项对你和项目最有意义的任务。如果这个选择有问题，那么团队中其他真正表现得像猪的成员会站出来给出意见。

- 细节会后讨论
  每日 Scrum 例会的目标是定位问题，而不是解决问题。如果在一两分钟的讨论后无法解决问题，那么请另外安排一个后续会议，自认与这个问题有关系的人可以参会。很多这种后续会议的内容都关乎哪些人要负责哪些任务。这就是团队自组织的方式：大部分任务都可以让大家自己给自己分配，但是有一些任务还需要讨论。只有通过每日 Scrum 例会的“检查”环节才能认清哪些问题是可以自分配，还有哪些问题是需要讨论的。

- 轮流先行
  没有谁充当进度的“守护者”，也没有人在项目中比别人重要。显然，有一些开发人员的专业技能比其他开发人员更高超，但是任何一个人都可能有好的想法。如果团队中有初级员工出了好主意，请不要因为他不是顶尖程序员而忽视他的想法。他可能会发现任务安排中存在的严重问题，整个团队都需要处理这个问题。为了让每个人都能听取其他人的好想法，每天的 Scrum 例会可以由不同的人起头。

- 不要当作例行公事
  我们每天都要开这些会（有些 Scrum 团队甚至把例会称为“仪式”），所有人都需要参加会议，并且参与每一个步骤。例会很容易将每个人都要回答的三个问题（从上一次例会到现在我都干了些什么？到下一次例会前我要干什么？什么事情阻挡了我的进度？）当作是例行公事，大家只顾着回答问题，而忘记了这么做的初衷。随着时间的推移，例行公事会模糊掉工作实质，因为人们会慢慢产生敷衍行为。这三个问题是每日 Scrum 例会的核心部分，因为团队每天都需要检查这些内容，这样才能尽早地发现问题。比如说，要找到因为某个人要承担太多任务而导致的瓶颈的最有效方法，就是让每个人回答关于阻碍进度的问题，因为第一个会被瓶颈阻碍进度的人肯定比其他人更早发现问题。

- 所有人都要参与
  所有人包括测试工程师、业务分析师以及团队中其他所有人，产品所有者也算在里面。所有人都要真正投入项目。产品所有者的工作非常重要，因为他要让所有人都知道积压工作表中有哪些任务对用户和公司来说最有价值，他要让所有人了解最新的状态。团队对于自己要交付的价值越了解，他们就能越准确地满足用户的目标。产品所有者还要与团队其他成员一起回答那三个问题，因为团队很容易忘记他在项目中也承担了重要的工作，而他的答案可以帮助团队其他成员理解他的工作。（事实证明，与用户交谈，理解用户的业务需求，以及管理积压工作表的确是需要全力投入的工作，如果开发人员能切身了解这些，那么他们就能更尊重产品所有者的工作。）

- 不要开成最新状态汇报会

  典型的状态汇报会是每周例行公事的一种典型。我们早已习惯了这种会议，所以很少想到要质疑，甚至都不会多想。状态汇报会应该达成两个目的：一个是让团队里的每一位成员都获得最新信息，另一个是让管理层获得最新信息。但是对于大多数团队来说，这个会议只是一种单向沟通，即单个团队成员与项目经理之间的二人谈话。为了避免这种状况，可以尝试确保每日 Scrum 例会中的每个人都在认真倾听。（这意味着不能查看电子邮件，不能玩手机，甚至不能做与工作相关的事情！）当团队成员开始把每日 Scrum 例会看作提早发现问题，避免走错路浪费开发时间的方式时，大家就会觉得这个例会并没有官僚主义色彩。他们会知道，这是一种以开发人员为中心的实践，可以帮助大家开发出更好的代码。

- 检查每一项任务

  寻找障碍的时候，不要只盯着手头正在做的事情，而要检查“待处理”栏中的每一个条目，往后看几步，看看是否有存在问题的可能。如果发现了潜在的问题，最好现在就把这个问题拎出来与团队一起讨论，把障碍消除，而不是默默地留着问题直到后面爆发。这也是任务检查需要团队每个人都互相信任的原因。如果有人有意或无意地没有准确描述他正在做以及计划做的事情，那么团队就可能会错过一个潜在的障碍，而这个障碍如果没有及早移出，后面可能会导致更严重的问题。

- 计划需要则改变

  这是“可见 – 检查 – 调整”周期中的“调整”部分，也是自组织团队的关键工作。团队在每日 Scrum 例会中发现了一个障碍，然后在后续会议中发现他们有一个错误的估算，无法交付一项已经承诺的重要功能。那么继续坚持已经知道不可能实现的计划有意义吗？当然没有。积压工作表和任务板必须反映项目的真实情况，如果发现了一个问题，那么整个团队都必须共同修正积压工作表和任务板。这就是产品所有者作为猪的方便之处，因为他可以立即开始调整其他人的预期。只要记住，不要管人们现在发现了计划的变化之后会有多糟糕的反应。如果你现在不告诉他们，他们日后的反应会更糟糕，他们迟早会发现的。

要点回顾

- 在每日 Scrum 例会中，每一位团队成员都要回答三个问题：从上一次例会到现在我都干了些什么？到下一次例会前我要干什么？什么事情阻挡了我的进度？
- 团队每天都通过这些问题集体检查项目的计划，并且针对项目中发生的变化进行调整。在整个项目中，这些问题的答案给团队提供了持续的反馈。
- Scrum 团队在最终责任时刻作出决策，这样可以保留灵活的选择权，更容易针对变化进行调整。
- 每日 Scrum 例会是整个团队的事情，并不只限于 Scrum 主管和产品所有者，所有人都要平等参会。

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试采用敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

4.6　第3幕：将冲刺计划写到墙上

Eric、Roger 和 Avi 吃了一顿漫长的午饭，讨论如何用好每日 Scrum 例会。Roger 有了一个主意。在接下来的一次每日 Scrum 例会中，他要求一名初级开发人员在会议一开始回答上述三个问题。当她询问 Roger 她的新任务是什么，他保持沉默。在半分钟的时间里没有人说话，局面变得让人感到有点不舒服。正当 Roger 开始怀疑这么做是否真的合适的时候，有一名更资深一点的开发人员开口了。在与团队里其他一些人进行简短沟通之后，这名初级开发人员准确地知道了接下来要做什么。她从任务板的“待处理”栏中取出了一张索引卡作为她的任务，把名字写在上面，然后放到了“处理中”栏里。

之后，这次每日 Scrum 例会就进展得非常顺利了。似乎这样一次讨论就让整个团队开窍了。成员都开始讨论其他人的任务，最终只需要安排两个后续讨论来决定谁来干哪件事情。Roger 很欣喜地发现有一个后续会议甚至不需要他参加。他要领导的后续会议与某个开发人员有关。这名开发人员上周完成了一项任务的 95%，结果发现有一个严重的障碍阻挡了进度，他需要别人的帮助，但是不敢提出请求，因为他不想浪费团队其他成员的时间（也有可能是因为害怕丢脸）。

几轮每日 Scrum 例会之后，Eric 指出整个团队开始一起工作了，Roger 真心感到高兴。在接下来的一周里，他真切地体会到大家终于领悟到了自组织团队背后的思想，那就是作为一个团队在一起工作。每一天，整个团队都会共同决定下一天的工作，并且互相帮助，解决问题。他们以一个团队的身份通过每日 Scrum 例会，对每天实际做的事情进行日常审查，保证开发过程正确。

一切看上去都进展得非常顺利，直到这一轮冲刺结束。与之前的 6 轮冲刺一样，团队在这一轮冲刺结束后也交出了一份可以工作的新版本软件。

这一切却成了一场灾难。

Avi 参加完下一轮利益干系人的会议之后非常沮丧地回来了。他本来期望所有客户经理都对新版本的 Lolleaderz.com 成就编辑器感到兴奋，这个成就编辑器允许用户创建自己的成就并分享到社交网站。此外，开发团队还更新了条幅广告的功能，允许每一位客户经理为自己的每一位客户设置自定义页面，通过页面展示最新的页面浏览信息和广告开销信息。

然而实际情况则是：大部分客户经理都感到困惑，并且被新功能吓了一跳。他们觉得没有人告诉他们会有这么多东西发生变化。突然间，每一位客户经理都收到了大量询问这些新功能的客户语音邮件。在过去，他们可以根据 Roger 的进度计划充分准备推销新功能。而现在，一切都变化太快，他们感觉没有时间跟上步伐了。

Avi 这边还有更糟糕的消息。有一些利益干系人要求退回到老的计划表，还想知道开发团队是否能在下一个季度之前暂时不要发布任何新功能。公司听上去想要他们完全放弃使用 Scrum 并回退到老的瀑布式流程。太糟糕了！

有那么糟糕吗？ Eric 也听到了 Roger 听到的消息，但是他并没有意志消沉，相反，他看上去异常乐观。你觉得他为什么会觉得乐观呢？

4.7 冲刺、计划和回顾会议

对于有些项目来说，冲刺计划很简单，就好像终于要开发积压工作表中人们很早就让你开发的东西。如果有一项功能是你的用户一直在要求的，然后你把这项功能设置为最高优先级，这样的计划很轻松就完成了。这种情况下，冲刺计划该做什么就像常识。

但有时候，冲刺计划没那么简单。通常情况下，冲刺的计划要求你以及整个团队以一种从未采用过的方式去思考用户的需要和价值。当人们说 Scrum 很难的时候，他们通常就在说这种情况。

幸运的是，真正的 Scrum 团队有一种不是太神秘的武器可以应对这种问题：产品所有者。当产品所有者真正花时间去理解利益干系人的需求以及怎样给他们带来价值的时候，他就可以帮助团队在每一轮冲刺里最先解决公司需求的问题。通过让团队能看见价值，并且帮助团队在每一轮冲刺时制订一个新计划交付这个价值，他可以帮助整个团队从简单的增量式流程切换到真正的迭代式流程。如果团队可以在每一轮冲刺结束的时候召开有效的回顾会议，那么产品所有者就可以把团队收获的教训带回到公司，让所有人的期待与团队实际交付的价值保持一致。

4.7.1 迭代式与增量式

每一轮冲刺结束时交付的新版本价值何在？

如果你规划的是有时限的冲刺，那么请严格遵守时间限定，当时间用完的时候，整个团队就停止所有的工作。这样团队可以在每一轮冲刺结束的时候交付可工作的软件，你也能从这种模式中获得大量收益。你会得到一个例行检查点，由此把控软件的质量。你的产品所有者、用户和其他利益干系人可以看到功能完整的版本，这样你们都可以看到开发的这些特性是怎样组合到一起的。这种模式可以显著降低集成的风险，因为团队不需要等到项目快要结束的时候再去集成由不同的人开发的特性，结果却发现这些特性不能很好地在一起工作。

尝试通过思想实验来帮助理解集成的问题。假设你的团队里有两名成员负责完成一个程序中的两项不同的功能，这两个功能都会把用户当前的工作数据保存到文件中，但是他们采用了不同的方式实现。你能想象这些功能有多少种互相冲突的可能吗？下面通过几个例子抛砖引玉：一项功能可能使用了一个保存图标，而另一项功能使用的是文件菜单；两项功能采用了完全不兼容的方式访问共享资源；两项功能利用不兼容的格式保存文件；这两项功能有可能会覆盖应用程序管理的共享数据；此外还有很多其他各种各样的集成问题。你还能想出集成过程中可能出现的其他问题吗？如果你从事过多年的软件开发工作，可能都不需要想象，在你的实际开发中就有可能多次遇到类似的问题。

与等到项目结束的时候相比，在每一轮冲刺结束的时候把所有开发的内容整合在一起，可以帮助团队认识到这些问题，甚至可以避免很多类似的问题。这种工作方式还有另外的好处：沟通更有效，利益干系人的参与感更强，项目的状态也更容易度量。当团队把项目分解为多个阶段的时候，这种开发方式称为*增量式开发*（incremental development）。Scrum 冲刺就是一种把项目分解为增量的方法，因此 Scrum 是一种增量方法。

不过 Scrum 远不止此。Scrum 冲刺的意义不仅仅在于在固定时间安排内交付出可工作的软件。Scrum 的意义还在于理解软件能带来的价值，准确地理解如何交付这些价值，并且在找到了能交付更大价值的方法时改变开发过程。像 Scrum 这样的方法和流程以这种方式工作的时候，这种开发方式称为*迭代式开发*（iterative development）。因此，Scrum 既是一种增量式方法，也是一种迭代式方法。

Mike Cohn 在他的优秀书籍《用户故事与敏捷方法》中给出了一个非常好的解释，阐述了迭代式方法与增量式方法之间的关键区别。

> *迭代式流程是一种通过持续精炼而不断取得进步的流程。开发团队首先给出一个系统的初步版本，大家都清楚这个版本在某些（甚至很多）方面并不完善或是功能很弱。然后他们迭代性地改善这些方面，直到产品达到令人满意的状态。在每一轮迭代中，更多的细节被加入到软件中，因此软件也一步步地得到了改进。*
>
> *增量式流程是一种把软件分成多个部分开发和交付的流程。每一部分（或称为每一份增量）都代表了一组完整的功能子集。一份增量既可以小也可以大，既可以表示一个系统在小终端上的一个登录界面，也可以表示一组高度灵活的数据管理界面。每一份增量都要有完整的代码和测试。对一份增量的常见预期是这份增量的工作在事后不需要返工。*[11]

对于首先给出一个系统的初步版本，并意识到这个版本在某些方面并不完善或功能很弱，然后迭代式地改进这些方面的开发方式，我们已经有一个术语可以描述："可见 – 检查 – 调整"周期。实施增量式开发的 Scrum 团队也会采用他们已经在每日 Scrum 例会中使用的"可见 – 检查 – 调整"周期，并且把这种方法完整地应用于一个项目的整体。这对应的就是冲刺计划、管理冲刺积压工作表，以及召开回顾会议。

这也就是为什么产品所有者对于 Scrum 团队来说如此重要的原因，同样也是为什么 Scrum 中会有产品所有者这样一个独立的角色的原因。产品所有者的工作包括以下这些。

- 理解公司最强烈的需求是什么，并把相关的信息带给团队。
- 理解团队可以交付哪些软件功能。
- 判断哪些功能对于公司的价值最高，哪些功能的价值较低。
- 与团队一起合作，判断哪些功能更容易开发，哪些功能更难开发。
- 利用价值、难度、不确定性和复杂性等信息帮助团队选择每一轮冲刺中要开发的功能。
- 将上述信息传达给公司其他人，这样大家可以知道应该为下一版本的软件做什么准备。

注 11：《用户故事与敏捷方法》，Mike Cohn 著。

4.7.2 冲刺成也在于产品所有者，败也在于产品所有者

产品所有者在项目中有非常明确的职责。产品所有者要对产品积压工作表负责，负责从中选择出最高优先级的条目，然后要求团队在冲刺中完成这些条目。在冲刺计划过程中，他会与团队一起决策哪些条目要进入冲刺积压工作表（这个积压工作表由团队集体负责），并且代表公司认可团队已经开发完成的条目。这意味着产品所有者有很高的权威，而没有权力作出这些决策的产品所有者（或者说有权力，但是不敢作出这些决策的人）并不适合这个角色。这同样意味着产品所有者对于公司的价值有很好的感觉。产品所有者可以与其他人讨论判断哪些条目的价值更高或更低，但是最后，他要负责对产品积压工作表中的所有条目做好优先级排序。

这也是为什么在冲刺一开始的时候，整个团队和产品所有者对于每一项条目的意义要有一致的看法是如此重要。当大家共同计划出了冲刺积压工作表之后，他们都需要一致地理解每一项的"完成"是什么意义，即要真正地完成。积压工作表条目真正"完成"意味着这一条功能已经被产品所有者接受并且可以交付给公司其他人。如果每一项条目都没有一个明确的、无歧义的对于"完成"意义的定义，那么在开发过程中会产生很多困惑，而且在冲刺结束的时候几乎一定会爆发出争执。但是如果每个人都对"完成"有明确的概念，那么团队在任何时刻都可以很清楚地了解自己处在冲刺的哪一个阶段。

冲刺是有时限的，通常为 30 天（还有一些 Scrum 团队会选择更短的冲刺，例如两周或三周）。当冲刺结束的时候，所有已经"完成"的条目都得到了产品所有者的认可。所有没有"完成"的条目都回到产品积压工作表中，即使团队在这些条目上做了大量工作（甚至几乎把时间都花在这些条目上面）。这并不是说团队要把这些工作放弃，删除原代码，完全恢复原状。这只是说在他们真正"完成"这些条目并且被产品所有者认可之后，才会承认他们完成了这些工作。

这种方式是很有价值的，因为这可以确保团队不会给用户带来一种他们应该交付的价值并没有真正交付的印象。对于所作出的承诺，保守一点总是更好。冲刺审查是整个团队真正面对用户和利益干系人演示过去 30 天所做工作的唯一机会。如果他们没有完成所有的承诺，那么团队必须坦诚地向用户解释他们交付了什么以及没交付什么。这是一种帮助大家真正感受到集体承诺的有力工具。同样，对于用户和利益干系人来说，这也是一个向团队询问问题的机会，这种交互帮助大家更好地沟通，并更好地理解什么是真正有价值的，他们可以开始构建真心的信任感。大家像这样碰头聊软件越多，那么信任感就越强烈，团队在未来开发软件的时候感受到的自由程度也越高。但是，尽管用户和利益干系人都在场与团队沟通，产品所有者还是要代表公司接受团队所作的工作。

在（希望如此）很罕见的情况下，产品所有者和团队发现冲刺的计划非常糟糕，或者发现了有一些非常紧急的变化不能等到冲刺结束。在这种情况下，产品所有者有权中止冲刺，暂停所有工作，把冲刺积压工作表中的所有事项移到产品积压工作表中。这种情况应该是非常罕见的，因为这会影响团队和用户及利益干系人之间好不容易建立好的信任。

4.7.3 可见性和价值观

想想你的工作动机。你在工作的时候，有多少次脑子里会出现下面这些想法。

- “使用了这项技术能让我的简历更漂亮”（开发人员）
- “如果我在这个项目中证明了自己，我就可以扩张我的团队”（团队主管）
- “赶上了这个重要的截止时间可以让我得到那个晋升机会”（项目经理）
- “如果我得到那个大客户，我就可以得到一笔很大的奖励”（客户经理、产品所有者或其他利益干系人等）

我们都会想这些事情，这是很正常的。

我们每个人都会有自己的兴趣所在，这是非常正常的。但是鼓励每个人的个人兴趣并不是聚拢团队的最有效方法。换个角度，如果每个人都朝着某个单一的目标努力，那么他们完成的工作会比他们每个人都自己孤军奋战完成的工作多得多。因此，尽管我们每个人对于能通过工作得到的东西有最基本的需求，但是如果我们都主要关注这些需求，那么我们能做到的比我们以一个团队合作能做到的事情要少很多。

下面是一个个人兴趣毁掉一个项目的例子：假设有一位团队成员会把他无趣或讨厌的工作丢给别人，而接受任务的人通常是团队中更初级的成员。我们大都见过这种场景。比如说，资深的开发人员经常会非常忙于开发新的软件，所以没有时间修 bug。而并非巧合的是，对于他们来说，开发新的软件的乐趣也要大得多，特别是在可以顺便学习新技术的情况下更是如此。对于会这样做的开发人员来说，要是有一个由初级开发人员组成的维护团队负责修所有的 bug 就很好了（这个团队还有可能在开发人员工资更低的城市里）。这种事情非常普遍，所以有很多公司就会按照这种方式组织团队：有一个由资深开发人员组成的“A 队”负责开发新功能，有一个由经验少的开发人员组成的维护团队负责修复 bug 并为已经发布的软件创建补丁。

对于那些喜欢“写了就丢”的编码风格的资深开发人员来说，这种工作方式很不错、很有趣、也很令人满意，因为他相信所有的 bug 一定都会被某个人修好，他完全不用去想这些问题。但是，尽管这种方式短期内对于团队中的一个人来说是很快乐的，但是对于要长期运行的团队来说，是一种非常低效的方式。几乎在任何情况下，引入一个 bug 的人是修复这个 bug 的最佳人选。他已经了解代码的所有细节，因为代码就是他写的，代码的风格完全符合他的直觉。而把这个 bug 交给其他人去修还需要沟通成本；有时候一封电子邮件就可以沟通，但是更多的情况下还需要额外的文档（例如 bug 报告，或更新的详细说明书）。另外，负责修这个 bug 的人还要花时间去读懂代码，这样才能了解对应的代码是做什么的。对于引入这个 bug 的人来说，可能几分钟就能修好，但是对于其他人来说可能需要数小时甚至数天的时间，特别是负责修 bug 的人还是更初级、经验更少的员工。

“把无聊工作外包”的风格在 Scrum 团队中很少看到，因为 Scrum 团队中的每一位成员都是恪守承诺的。如果一个团队中的资深开发人员能在问题还处于新鲜状态的时候用几分钟的时间完成“无聊的工作”，而不是让更初级的团队成员在事后耗费数小时的时间，那么这可以成为让其他团队感到的无法理解的 Scrum“超高生产率”和“惊人成果”的另一个原因。

任何团队（甚至包括非敏捷团队）都可以设置规则不允许资深团队成员把“无聊的”维护性工作丢给初级员工，以此来维护上述生产率。但是真正的 Scrum 团队并不需要为此设立规则，也不需要对任何不同的情形设置不同的规则。原因是在 Scrum 团队中，每个人都真

心感觉对项目的每一方面负责。如果资深团队成员这么想的话，他肯定把这些任务丢给其他人，而事实上，他可能根本想都不会这么想。他是完成修复工作的合理人选，因此为了把事情做对,他会觉得这件事情是最重要的事情（就像 CEO 取咖啡的那个例子一样）[12]。修复 bug 和其他维护性的任务也应该加到冲刺积压工作表中，每天 Scrum 例会都要审核这些任务。像其他任何任务一样，这些任务也要由正确的人在正确的时间内完成。（修复 bug 的最后责任时刻可能就是现在，因为这个问题在开发人员的大脑中仍记忆犹新。）

在真正的 Scrum 团队中，每个人都有责任感，这个责任感不仅在于他们开发的代码，还在于积压工作表，以及所有人为交付可用软件所做的工作。冲刺积压工作表反映的是所有人（甚至包括最初级的开发人员）的真正承诺，这感觉就是他们对用户作出的承诺。这也是 Ken Schwaber 在本章开头引言中"集体承诺"的真正含义所在：团队中所有成员都要对积压工作表负责，并感受到给用户交付他们能做到的最有价值的软件是个人的责任，这样的软件要包含所有功能，而不只他们正在开发的那些功能。

团队成员都对项目有责任感，所以每个人（从最初级的开发人员到高级技术主管到 Scrum 主管和产品所有者）都会自愿地承担起无趣烦人的任务，因为他们都真心对项目感到关心。你怎么样才能找到这种感觉呢？

令人鼓舞的目标，激发团队中所有成员的积极性

你有没有自愿地完成工作？有没有为一个开源项目做过贡献？有没有加入过某个俱乐部、某个业余体育队、某个摇滚乐队、或某个教堂唱诗班？回想一下你上一次加入工作或家庭之外的组织。你为什么要这么做？

你加入了一个组织，可能为这个组织你投入了很多精力，因为你关心这个组织要做的事情是什么。如果你加入了一个选民运动，那么你关心的是要让人们参加到选举中来。如果你在一个足球队里，那么你关心的是赢比赛（也有可能关心的是踢得好）。那么工作不能也像这样吗？

我们所有人都是自我驱动的。至少，我们为钱工作。如果你的职位不给你付钱了，你就不会出勤。我们有账单要付，还有很多人有家庭要抚养。因此，给我们付钱（并且有一个干净安全的办公环境，给我们可工作的时间，以及所有构成办公环境的其他东西）就足以让我们在工作时段内坐在屋里的办公桌前工作。

但是，这是否足以让我们真正关心开发优秀的软件？

如果你曾经在一个没有真正有积极性的团队工作过，那么你肯定知道这个问题的答案是否定的。而事实上是，有很多人从来都没有在真正有积极性的团队工作过。如果有的话，你现在可能已经想到那个团队了，因为这段工作在你职业生涯中应该是体验最棒的。当所有人都开始关心开发伟大软件的时候，事情就会变得更好：人们沟通得更多，争论得更少(即使有争论，他们也会更热情，而且效率更高)，大家就这么把事情搞定了。

团队的热情可以被很多事情激发：得到机会使用新技术或进入他们想学习的领域、晋升的可能性、绩效奖励以及在家工作等。团队也会被负面事物激发：老板会生气，你会被老板

注 12：实际上，在真正的 Scrum 团队中的人读到这里可能会感到奇怪，甚至有可能会感到有一点愤怒，那就是看到居然会有资深的员工把任务"丢给"其他人去做，这种想法简直就是太诡异了。

怒吼（或者减薪，甚至被解雇！）。这些事情不管是正面的还是负面的，对人们的激励都代表了人们个人的兴趣，而不是针对团队本身这个整体的激励。

如果围绕着一个令人鼓舞的目标组织团队，那么这个团队就能得到有效的激励。在本书作者的另一本书《团队之美》中，项目管理作者、专家 Steve McConnell 给出了下面这样一个关于令人鼓舞目标的定义。

> 如果你在外面挖水槽，这件事情一点也不振奋人心。但是如果你挖水槽的目的是要保护自己的城池不被敌人攻击，那么这件事情就变得更加鼓舞人心了，尽管做的实际上是同一件事情。因此领导的真正职责就在于要用一种人们能够理解其价值的方式来表达需要做的事情。

几乎所有的软件都是由团队而不是个人开发的。要让一个团队有效地工作，这个团队就需要整体激励，而不是针对个人激励。激励团队最好的方式是采用令人鼓舞的目标，这样可以让团队一起合作，为一个大家都关心的更高目标而工作。

交付价值是一个非常有效的可以激励整个团队的奋斗目标。如果一个团队有一个大家都真心相信的奋斗目标，那么他们就会自己考虑如何实现这个目标（以及接受风险的自由），他们会尽力使用各种可能的工具解决实现这个目标途中的各种问题。

令人鼓舞的目标在乎的是价值，但是“价值”这个词本身看上去似乎很抽象。在敏捷团队中，价值有着非常真实具体的意义：软件只有能让用户生活更美好的时候才体现出价值。如果一名高级副总裁走到团队中这样对大家说：“由于你们过去几个月里的辛勤工作，我们第三季度的营收额提升了 0.024%。大家干得好！”会有什么样的后果？这种说法并不能真正激发大部分开发人员，即使是对于那些拥有股票期权的开发人员也无法激发热情。

从另一个角度看，如果同一个人走到团队中这样对大家说：“哇，我过去每天要花三个小时的时间来整理这些数字。你们的软件太好用了，我现在只要花 10 分钟就可以搞定所有事情。太感谢你们了！”对于大部分开发人员来说，这种说法更令人鼓舞。

开发人员（这里说的“开发人员”指的是敏捷团队中的所有成员，包括那些不直接写代码的成员）会因为满足了*技艺的自豪感*（pride of workmanship）而受到高度鼓舞。我们希望开发出有用、人们喜欢和爱护的软件。我们希望我们开发出来的软件能优质高效地完成任务。我们还希望我们开发的软件能尽可能地优秀，所以才会花这么多的时间去争论不同的设计、架构和技术。这些东西是团队真正关心的。让用户生活更美好（这是我们交付价值最直接的方式）是一种真实、真诚、鼓舞人心的目标。

这一条原则是敏捷宣言中原则列表中的头条，注意其中我们强调的词语（用黑体表示）。

> 最优先要做的是尽早、持续地交付**有价值的软件**，让客户满意。

这条原则的优先级最高的原因在于：给用户交付价值是鼓舞团队的最有效方法，这也是好的冲刺规划背后的驱动力。

4.7.4 计划并执行有效的Scrum冲刺

- 从积压工作表开始，即从用户的角度出发

  为什么我们在这一轮冲刺要交付某项具体的功能，而不是把这项功能推到下一轮冲刺中？因为我们整个团队在一起判断哪些功能对用户是最有价值的。这也是为什么产品所有者如此重要的原因，他的职责是理解用户，并且让团队中所有人了解用户对软件的最新需求。

- 对你能交付的东西要现实

  很多经理都有一种疯狂的想法，那就是如果不压迫开发人员，开发人员就会尽可能地偷懒少干活，以及给自己设定很晚的截止时间。在大部分团队里，事实与此相反。在现实中，开发人员往往会过于乐观，因此我们都经历过项目延期的情况，但是很少见到项目提早交付的情况。不要尝试在一个冲刺中塞入太多的特性了。（毕竟，用户只需要等到下一轮冲刺的时候拿到的可工作的软件就包含下一组功能了。）好的 Scrum 主管会帮助团队预估工作量，并判断哪些功能可以包含进来，哪些功能不能包含进来。

- 在有必要的情况下改变计划

  充分利用每日 Scrum 例会，了解团队是否能真正完成他们承诺要在冲刺中完成的工作。如果计划需要改变，那么团队有责任改变计划。如果大家都很清楚整个团队无法完成冲刺积压工作表中的所有工作，那么团队应该从冲刺积压工作表中转移一些条目（从价值最低的条目开始）到产品积压工作表中。Scrum 主管应该确保团队中所有人都了解这个变化。习惯于频繁看到新版可工作软件的用户通常情况下不会因为冲刺审查中没有看到他们期待的所有功能而感到不愉快，特别是在产品所有者已经很好地沟通了他们的期待的情况下。

- 让大家都关心价值

  有效的 Scrum 团队中的成员一般都理解用户真正需要的是什么，以及对用户有价值的东西是什么。做到这一点的唯一方式就是团队中的所有人都理解开发的软件能为使用这些软件的人带来什么。如何通过软件让用户的生活更方便？软件可以让他们做到哪些他们以前无法做到的事情？软件可以为他们节省多少时间和精力？这些事情对敏捷开发人员来说都很重要。你和他们沟通得越多，了解他们关注的事情越多，那么你开发出来的软件就越好。

要点回顾

- Scrum 既是增量式的也是迭代式的，增量是在于 Scrum 把任务分解为连续的冲刺，迭代是在于团队在每一轮新的冲刺中都会根据项目中发生的变化进行调整。
- 当真正的 Scrum 团队说他们被“交付价值”所激励的时候，他们的意思是说他们最重要的目标就是要开发出改善用户生活的软件。
- 产品所有者的职责是帮助团队理解他们的用户、理解用户所做的工作以及理解用户如何使用软件，通过这种方式让团队能一直受到交付价值的激励。

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试采用敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

4.8 第4幕：尽力之后

Roger 和 Avi 回来跟 Eric 说了那个“地狱般的利益干系人会议”，Eric 居然表现得很乐观，这让他们无法理解。几天后，他们三人提早下班，来到附近一家餐馆，准备探讨一番。

Eric 首先开口：“你们觉得客户经理为什么会不舒服呢？”

Roger 和 Avi 都想不出合适的答案。Roger 说起他们为敏捷所做的诸多工作。在他的脑海中，敏捷意味着针对变化的功能请求持续调整，并及时开发全新产品。Avi 觉得他为融入团队做了很多努力，而且能够从所有的客户经理那里持续获得一大堆很棒的想法。他们都觉得自己准确地完成了利益干系人要求的工作。“你看，这里有每一位客户经理要求我们实现的功能，我们都交付了，”Avi 说道，“他们怎么可能会对这些功能感到不满呢？”

现在，作为一名敏捷教练，Eric 要开始解释为什么客户经理会不满了。他对他看到的现象表示满意，尽管项目看上去遇到了麻烦。他之前解释过，他们的团队过去就像一艘巨大的邮轮，转起弯来非常笨拙。现在他们更像一组高度协调的快艇组成的舰队。尽管需要大量的沟通，但是可以转急弯。Avi 之前过度劳累，而且同团队关系紧张，到现在他成为了团队真正的一员，并且为团队与用户有效搭建了桥梁。

通过帮助团队实施有效的每日 Scrum 例会，Roger 让团队有能力根据变化的优先级实现自组织。但是出现了一个新问题，Eric 解释说很多迈过了成为自组织团队第一道坎的团队都遇到过这样的问题。一个新的自组织团队现在有能力极为高效地转变方向。深入团队的产品所有者现在也发现有权设置这个方向。

Eric 打了一个很好的比方：“你有没有看到过消防员训练使用高压水枪？灭火的时候，看上去他们只是拿高压水枪指向火焰。但是他们需要耗费数周数月的时间去学习如何高效地移动它，还要互相沟通，这样大家才能朝着同一个方向移动。你的团队之前是花园的浇花软管，而现在他们是高压水枪。你们两个现在在尽力抓住水枪，指向正确的方向。现在你们要学会如何一起移动，让水喷向火焰。”

常见问题

如何处理项目计划中任务之间的依赖关系？

与原先处理方式一样，就是与团队一起讨论，尝试找出这些依赖关系。

对于熟悉命令 - 控制式项目管理，习惯事先计划好全部需求，并在项目初期画出巨大甘特图的项目经理来说，这是一个非常普遍的问题。自组织团队（每一位团队成员自己决定下一个任务要做什么）的思想听上去不现实。通常情况下，这是因为项目经理耗费了很多时间和精力识别任务之间的依赖关系。项目管理相关的课本（甚至包括本书作者写的一些课本）都有章节讲解不同类型的依赖关系（例如结束—开始、开始—开始等），以及如何识别这些依赖关系并且在项目一开始进行记录。因此，问出如何在自组织团队中进行依赖关系分析的问题也很自然了。

为什么项目经理在项目开始的时候需要这些依赖关系？因为他们要定义项目的工作，将其分解为任务，对这些任务排序，安排资源，然后安排项目进度表。对项目排序的唯一方式就是找出这些任务之间的依赖关系。

假设一名团队成员正在完成一个大计划中的一项任务，在他工作的时候发现这项任务依赖另一项任务。啊！现在计划中所有下游的任务都需要往后推，因为任务序列并没有考虑到这个之前没有发现的依赖关系。这造成了层叠式延误，这是项目计划必须变更的一个最常见原因。更糟糕的是，如果团队正努力在某个固定的时间内完成工作，那么项目经理在后期需要作出艰难的决定，然后就削减项目进行痛苦的谈话。难怪项目经理会如此痴迷于依赖关系！未发现的依赖常常导致精良的计划变成一团糟。因此，完整的依赖分析真的会在团队面临的风险上给项目经理带来虚假的安全感，最终导致项目延迟。这种延迟似乎几乎总是在最糟糕的时刻发生。团队也有一种虚假的安全感，因此一旦项目计划经过了审核并分发给整个团队之，大家就不会再花多少时间考虑依赖关系了。

自组织的团队也会发现依赖关系，但是成员有更好的方式去应对。他们在最后职责时刻处理依赖关系，这个时候他们已经获得了足够多的任务信息，可以执行更为全面的分析。

理论上听起来都不错，可是在真实的项目中真的有用吗？

回想一下第 3 章中低效 Scrum 例会的例子，在这个例子中，命令 – 控制式项目经理负责每天召开例会，并且给每一位团队成员分配任务。为了能正确地分配任务，项目经理需要知道会议中所有任务之间的依赖。把这个例子与真正的自组织团队作比较。当一名团队成员给自己分配任务的时候，整个团队都在看着，并且在会议期间积极地寻找依赖关系，还会思考之前会议中提到的任务与以后会议中提到的任务的依赖关系。因此，团队成员要在例会中回答第三个问题（也就是和障碍相关的问题）。每一个障碍都是整个团队要尝试发觉的依赖关系，每天每一位团队成员都花时间来寻找这些依赖关系，所以他们能找到。命令 – 控制式项目中的层叠延期就这样避免了。

为什么这样会有用呢？当命令 – 控制式项目经理在项目计划阶段进行依赖关系分析的时候，他依靠的是专家的判断力。这实际上是指项目经理同团队成员沟通，并依靠他们的专业知识整理依赖。自组织的团队每天在开 Scrum 例会也会靠专家的判断力去检查整个计划。但是成员现在有了更丰富的信息，因为他们每天都会在一起以整个团队的身份做这件事情。随着项目的推进灵活计划可以让信息传达更加顺畅，利于团队作出更好的决策。自组织团队更容易找到关键的依赖关系，因为成员可以在项目进行的过程中寻找依赖。相对而言，命令 – 控制式团队的更加不透明，拥有的信息更贫乏，因为他们在工作真正开始前数周或数月进行分析。信息更透明，即时依赖性分析更到位，这是 Scrum 团队生产力的一个重要来源。

我有点不太喜欢这种“最后责任时刻”的想法。就算计划真的要变化，那不也是提前做计划不是更好吗？

项目经理经常喜欢引用美国前总统艾森豪威尔将军的话：“在准备战斗的时候，我总是发现计划一文不值，但是计划是不可或缺的。”坐下来与团队一起计划项目，这样可以让大家认真思考工作细节，以及把工作做好要解决哪些具体问题。即使项目经理和团队给出的预估并不完美，在一轮好的计划过程之后，团队还是可以更了解项目，大家都可以为项目更好地做准备。当变化发生的时候，团队能捕捉到变化，然后从变化中吸取教训，在未来避免问题。那么为什么不事先做好所有的计划，获得这些好处呢？计划不是可以按照需要改变吗？

在最后责任时刻做计划的团队不仅能得到上述所有好处，还有很多其他好处。这并不是说团队事先不要做任何计划。事实上，敏捷团队事先也是会做计划的。区别在于，早期计划的是大事，即计划产品积压工作表中的条目。产品积压工作表包含的条目（通常是用户故事）是产品所有者和公司中其他非技术人员也能理解的。在第 1 轮冲刺开始之前，团队和产品所有者需要对冲刺积压工作表达成一致，这就要求产品积压工作表更长，要带有对故事点的预计。所以他们必须在第 1 轮冲刺开始之前准备好产品积压工作表。团队在项目开始的时候确实还是需要做大量的计划工作！

能不能给出一个 Scrum 冲刺做计划的真实例子？

当然可以。下面是我们在真实生活中见到的例子。假设你在一个传统的项目团队中开发新功能，还有一些性能调校的任务。如果你在对这个项目做计划，你会希望团队最先做什么？大部分团队最终的计划都是先开发新功能，把性能调校的任务放在项目最后。如果你是一名开发人员，你很有可能会觉得新功能开发要有趣得多，而追踪和修复性能问题则更为困难和令人沮丧，这种工作自然会排到计划的末尾。

但是，如果用户真的非常关心性能怎么办？要是软件性能问题已经妨碍了用户的工作，软件更新更快比功能更多重要，应该怎么办？如果是这样的话，团队应该把性能调校安排在最高优先级。在这种情况下，如果产品所有者每周都花时间告诉团队积压工作表中哪些功能更重要，他就更有机会让性能改进的工作最先完成。

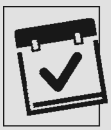
再跟我解释一下，这与最后责任时刻做决策有什么关系？

如果团队事先完成所有的计划，那么成员必须一开始就知道性能问题对于用户最重要，不然就不会把性能调优放在新功能之前。如果产品所有者没有养成习惯，没能持续地向团队灌输用户价值，而且团队成员也没有听取这些内容的习惯，那么如果产品所有者在项目进行途中突然跑过来反馈说用户真的需要优先进行那些性能调优，这时候会发生什么？项目经理和团队很有可能会把这个请求挡回去，因为这要求重大变化，会破坏整个计划。他们还会私底下抱怨与他们合作的业务负责人总是改变想法，然后期待不怎么发生变化的项目。

这样的团队也能搞定需求变更。但是处理这些变更需要耗费大量精力，而且很有可能给项目带来巨大变化。在现实世界里，几乎所有用户都要处理不断变化带来的问题，这就是业务的本质，也是生活的本质。团队把变化看作例外是不切实际的。这也就是敏捷团队喜欢说**接纳变化**的原因。

这种不灵活的“事先做好计划”的态度会把项目中的变化变成团队与产品所有者之间的谈判，而我们已经知道了敏捷团队珍重客户协作高于合同谈判。在谈判中，有人会赢，有人会输，所有人都要妥协。这并不是运营项目的高效方法，会损失很多生产力。

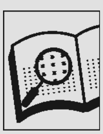
而从另一角度看，如果团队具有与客户协作的态度，所有人都协同工作，那么所有人都能赢，不会有输家，因为所有人都很自然地接受他们必须在一起工作的约束。在最后责任时刻作出计划的方式鼓励了这种态度，因为这种做法可以防止团队遇到随意提出的要求，导致谈判。这让团队对变化敞开了大门，也可以让大家在项目进行中为了满足变化的目标（与之相反的是做好工作计划，然后按照计划工作）而重新调整工作（也许并不容易，但是有可能做到）。同样，产品所有者也更容易持续地参与到团队中，实现这些目标。



现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 如果你加入的团队已经有了每日站立会议，那么让每个人都回答 Scrum 例会中需要回答的三个问题。
- 如果你加入的团队没有每日站立会议，那么看看有没有机会组织一个。
- 与团队一起讨论最后责任时刻。把你和你的团队那些可能因为作得太早而需要复审的决策列写下来。
- 与团队一起讨论可能会出现在产品积压工作表中的条目。还有哪些工作你们还没有做？能不能把这些条目列出来？



更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 《Scrum 敏捷项目管理》，Ken Schwaber 著：进一步了解自组织团队以及如何管理 Scrum 项目。
- 《敏捷估计与规划》，Mike Cohn 著：进一步了解 Scrum 项目计划。
- *The Scrum Guide*，可以从 http://www.scrum.org 下载：进一步了解 Scrum 的规则。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 采用 Scrum 最困难的一个步骤就是找到产品所有者。如果与你共事的团队想采用 Scrum，那么你要帮他们找一个有这种权威以及意愿的人，这个人要代表业务方作出决策。
- 很多敏捷教练发现他们带领的 Scrum 团队遇到问题的原因是他们随便在团队中选了一个人做产品所有者。帮助他们理解产品所有者需要有权代表业务方的利益接受或否决团队做出的功能。
- 团队召开的每日 Scrum 例会是否本质上还是状态汇报会？借此机会帮助团队明白命令 – 控制式团队与自组织团队的区别。
- 帮助 Scrum 主管理明白，他不负责告诉每一位团队成员每天要干什么，也不要跟踪他们是否完成了工作。让团队明白，Scrum 主管的责任是确保大家都遵循 Scrum 的规则，并为他们铲除障碍。

第5章

Scrum计划和集体承诺

> 每位开发人员都必须全心全意地投入自己所选择的工作中。整个团队应该有一种“同舟共济”的态度，每个人都应该认同团队的整体承诺。
>
> ——Mike Cohn，《用户故事与敏捷方法》

你已经学习了 Scrum 的机制，以及如何在团队中使用基本的 Scrum 模式协同工作。但是，Scrum 理论与团队真刀真枪地做软件项目开发之间有着很大的区别。如何安排你的 Scrum 团队才能成功？如何让团队中的每个人都为了共同的目标而努力？换句话说，你现在已经了解了 Scrum 的基本原则是自发组织和集体承诺，可是怎么让团队在现实生活中按照这些原则去做呢？

本章中，你会学到很多 Scrum 团队所使用的计划冲刺的方法。你会看到用户故事如何帮助你确切了解用户需求，并且将使用故事点和项目速度来估计你的团队在一次冲刺中能完成多少工作。同时，你将学习如何使用两个有价值的可视化工具（燃尽图和任务板）来让所有人保持同步。

你也会了解为什么这些做法和基本的 Scrum 模式不足以让团队实现超高效率或者取得惊人的成绩。我们会重温 Scrum 的价值，而你将学到如何评估团队和公司文化是否与这些价值相符，以及不相符应该怎么办。



图 5-1：对产品所有者来说，获取用户的真实需求非常困难

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试采用敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

5.1　第5幕：出乎意料

第二天，整个团队一起讨论如何能够协同行动从而让项目的进展更加易于预测。他们讨论了集体承诺，以及这个词的真正含义。Eric 让几个团队成员讲述他们看到软件真正被很多人使用的经历。讲述过程中大家都不住地点头和微笑，每个人似乎都认同：当他们开发的软件真正被使用的时候，是他们最爱自己的工作的时候。然后，Eric 又让大家说说软件完全没人使用的情形。有人提起了去年，这个团队花费四个月开发了一个用户跟踪和账户管理系统，结果一个高级副总裁在最后时刻决定弃之不用，转而从外面采购了一个系统。经过这件事后，两个员工递交了辞呈，所有人都感到不爽。一个曾经为了该项目而时常加班的资深开发人员略带怀疑地问：“怎么能够避免类似的情况发生呢？”

Roger 进行了解释："当用户使用我们的软件时，我们会感到高兴；反过来，如果我们的工作成果被用户抛弃，我们就会沮丧。所以，我们所需要的就是一种方法，保证只开发用户买账的软件。"

这之后，整个团队很快都表示认同了。Roger 解释了冲刺计划是怎么回事，以及 Avi 如何与团队和用户协作，以保证只有最有价值的特性才能进入冲刺积压工作表。到会议结束时，Avi、Roger 和 Eric 都觉得整个团队已经开始理解集体承诺的意义，并由衷地希望开发出用户真正感到有价值的软件。

会后，Eric、Roger 和 Avi 坐下来做总结。Roger 和 Avi 彼此表示祝贺，终于让整个团队想到一处了。可是他们再次对 Eric 的反应感到惊讶：Eric 似乎并不乐观，反而有些担忧。

Eric 说道："好，你们已经成功地让整个团队参与到冲刺计划中了。这当然很好，但是这并没有解决那个更为关键的问题：如何让正确的产品特性进入冲刺积压工作表？这个问题之所以更为关键，是因为如果你们无法做到仅让最有价值的特性进入冲刺积压工作表，那最终的结果跟现在的状况就不会有什么两样，用户会抱怨软件缺失了重要功能，却多了一堆他们不需要的特性。"

Roger 和 Avi 对 Eric 的话将信将疑，不过这丝怀疑在他们接下来与用户一同进行冲刺评审时就烟消云散了。在评审中，Avi 不无骄傲地打开了测试机上的 Lolleaderz.com 网站的最新版本，并开始演示他们最近开发的"新建成就"功能。这个功能可以让用户定义一组规则，他们的朋友则可以上传视频来达成这些"成就"。Avi 在新的成就编辑器里创建了一个名为"抓拍山羊"的成就：把"山羊"这个关键词从列表中拖拽出来，然后用新的工具箱控件指定要达成这个成就需要达到 500 个页面浏览；同时，他还用新开发的图片上传功能给这个成就加了一个装饰性的弹出动画。演示结束时，整个屋子一片寂静。

"呃！貌似这个工作量不小啊。"一个销售经理说，"那么，嗯……你们为什么开发这个功能呢？"

这句话对 Avi 和 Roger 来说无异于晴天霹雳！最后他们弄清楚了，用户需要的不过是一个简单的小功能来提名朋友的视频并给它旁边加个小小的星标。不知道怎么回事，开发团队把这个简单的需求搞成了一个全功能的成就编辑器，完整到它居然有自己的伪脚本语言和服务基础架构，销售经理完全不知道该如何把这个东西卖给客户。这当然是团队精力的巨大浪费，但更重要的是，现在他们的积压工作表里有一大堆应该实现的特性，却没有实现。

这跟想象中的情况差距太大了！

会后 Roger 和 Avi 回来找到 Eric，后者对此完全不感到意外。"那咱们怎么解决这个问题呢？"Roger 问。

Avi 说他已经尽量满足销售经理的要求了。他维护着积压工作表，按价值对它进行管理和组织，并把它拿给开发团队。他还能怎么改进呢？Roger 则完全没有头绪了，不过他清楚，如果他们当初没有做任何改变，将会得到相同的结果。但这一次，他们做了改变，给开发团队开了个誓师大会，让大家对 Scrum 燃起了热情，而现在的结果却让所有人失望。如果这种状况不能尽快改善，Roger 觉得他们可能将永远失去团队的信任了。

Eric 说："这里的技巧在于想用户之所想。你需要知道他们是如何使用软件的。更重要的是，在冲刺结束时，你需要一种方法来判断你的方向是否正确。如果能做到这一点，你所开发的软件就一直会有人用，而你的用户也会热爱你的产品。"

Roger 还是将信将疑，因为根据他的经验，用户很少知道他们真正需要什么，直到他们看到最终产品为止。而 Avi 则又开始对他的产品所有者角色产生了"脏活全是我的"的感觉。

Avi 问道："如何做到想用户之所想呢？"

5.2 用户故事、速度和普遍接受的Scrum实践

任何软件的用户或者利益干系人都痛恨不可预知性。即便你的团队由衷地共同努力开发出了他们所能开发出的最佳软件，如果冲刺结束时最终交付的产品与最初的许诺有出入，用户一样会失望。换句话说，仅仅开发出团队能力所及的最佳软件是不够的，你还需要保证用户在使用的时候不会有什么意外。

正面的意外（惊喜）有时可能跟负面的意外一样具有破坏性，如果这些惊喜抬高了用户的期望值从而认为你的团队总是能够超水平发挥的话。避免这些意外一般归结为两件事。第一，开发团队需要在冲刺一开始就预设好期望值；第二，随着冲刺向前推进，他们需要让所有人都了解每日例会中所发现的变动。这就是让用户和利益干系人参与每日例会的价值(但不是必须的)。这也是为什么他们应当只是列席，而不应进行干预。每日例会是用来让开发团队计划明天的工作的，而不是用来回答团队外部人员的问题的。并不是所有相关人员都能够参加每日例会，有时候可能一个都没有。这也不是什么问题。如果他们能够参加固然很好，但这不是必须的。只要产品所有者能够及时地把团队对计划的种种调整通知给各利益干系人，那么大家就能够保持步调一致。

但是，如果开发团队产出的软件没有价值，以上所说的一切都毫无意义了。

5.2.1 提升软件价值

回头再看一眼第 4 章开头 Ken Schwaber 的那句话。如果你不理解集体承诺，就不理解 Scrum。可是，"集体承诺"这个词的含义到底是什么？作为一个团队，你们到底共同承诺了什么？

集体承诺的意思是*努力尝试让你们的软件变得更加有用*。要让你的软件有用，你需要理解你的用户在干什么。你需要打心眼儿里在乎他们，希望帮助他们达到他们的目的，而且你需要把这一点作为第一要务：超过你对项目任何其他方面的关注。

上述原则是写入敏捷宣言的。试想一下，所谓构建"可用的软件"是什么意思，我们要如何定义"可用的"？让软件运行起来毫不费力；构建看起来"能用"的软件也不难，但当用户真正尝试用它来做实际的工作时，常常会被它逼疯；但构建真正意义上"可用"的软件意味着构建那种真正能够帮助用户完成工作的软件。而现存最有效的构建可用软件的方法就是客户协作。这也是我们强调软件的可用性超过完善的文档、强调客户协作超过合同谈判的原因。

在敏捷开发改变软件开发领域之前，开发出无用的软件是常事，我们有大量证据。如果你打开 20 世纪 90 年代末或 21 世纪初任何一本有关软件工程的教科书，几乎总能看到对 Standish Group 的 CHAOS 报告的引用。Standish Group 从 20 世纪 90 年代起开始做这个报告，针对的就是软件行业中的一种认知，而这种认知恰巧是正确的，即大量的软件项目都以失败告终（仅有约 1/3 的项目被认为是成功的 [1]）。他们的年度调查和研究反复发现一个现象，即项目团队感觉他们开发的软件中很多功能特性都没被用到。在 2002 年的研究 [2] 中，这个比例达到了匪夷所思的 64%（其中 45%“从未被使用”，19%“很少被使用”）。

这也许会令很多学术界的人感到震惊，但对于那时候的很多软件开发团队来讲，这是显而易见的。事实上，CHAOS 报告中所谓的“失败”实在太过普遍，很多开发团队都把这当作软件开发中天经地义的事情。开发团队把软件开发出来，扔给用户，然后期盼多少能有一些功能达到用户的要求。很多与用户之间的讨论（“争论”也许更合适）都以开发人员的这样一句话结束：“这不是一个 bug，这是一个功能。”开发人员的潜台词实际上是说，软件功能是按照他自己的意图开发的，而用户则需要调整自己从而适应软件。这么对待用户实在是太差劲了。

Roger 和 Avi 在他们的成就编辑器上遇到了麻烦，因为他们开发了一个过于复杂的工具去解决一个实际上很简单的问题。这是一个很普遍的问题，它经常导致开发团队最终交付的软件不能达到其原本可以达到的质量。事实上，这个问题已经普遍到有了一个特定称谓：“镏金”，即开发团队出于好的意图，给他们的软件添加一些用户从未要求过也完全不需要的额外特性。开发人员是出于好心，而他们也的确认为他们开发的东西很酷、很有价值。干劲十足的开发人员这么做是很自然的，不过这同时也是那 64% 无人问津的功能特性的主要来源。

大量无用特性的另外一个来源则是“扔给你拉倒”这种思维模式本身。由于这种模式下用户会感到与开发团队的沟通时间有限，所以在这有限的时间里，他们会试着提尽可能多的功能需求，这样就会导致他们提出一些软件交付后没什么实际价值的功能。

反过来，真正高效的敏捷团队很少遇到这样的问题。他们开发的功能很少会不被使用，其比例当然远远少于 2/3，甚至这些开发团队常常会认为 CHAOS 报告中的结论是不正确的。事实上，一些敏捷方法的传道者尖锐宣称 CHAOS 报告结果已经被“辟谣”的事情并不少见。那么，高效的敏捷团队到底做了什么，使得他们能够开发出真正有用的软件呢？

5.2.2 以用户故事构建用户真正会用到的功能

敏捷团队从“想用户之所想”入手，而且他们有一个非常有效的工具来做到这一点：用户故事。用户故事是一个看起来特别简单的工具。它就是对一个用户使用软件某一特定功能时的一个简明扼要的描述。大多数用户故事的长度大致为 1 ～ 4 句话，很多团队都会要求用户故事能写到一个 3 英寸 ×5 英寸（约 7.62cm×12.7cm）的卡片上。

一般编写用户故事时会采用一种类似填词游戏的方法：

注 1：“The Rise and Fall of the Chaos Report Figures”，J. Laurenz Eveleens 和 Chris Verhoef 合著，IEEE Software 第 27 卷 1 号（2010）。

注 2：*2002 CHAOS Report*，The Standish Group International Inc.，2002。

- 作为一个 < 用户类型 >，我要 < 要执行的某个具体的动作 > 以便 < 要达到的某种目的 >。

下面是 Lolleaderz.com 团队在开发成就系统之前可以用到的一个用户故事的例子。

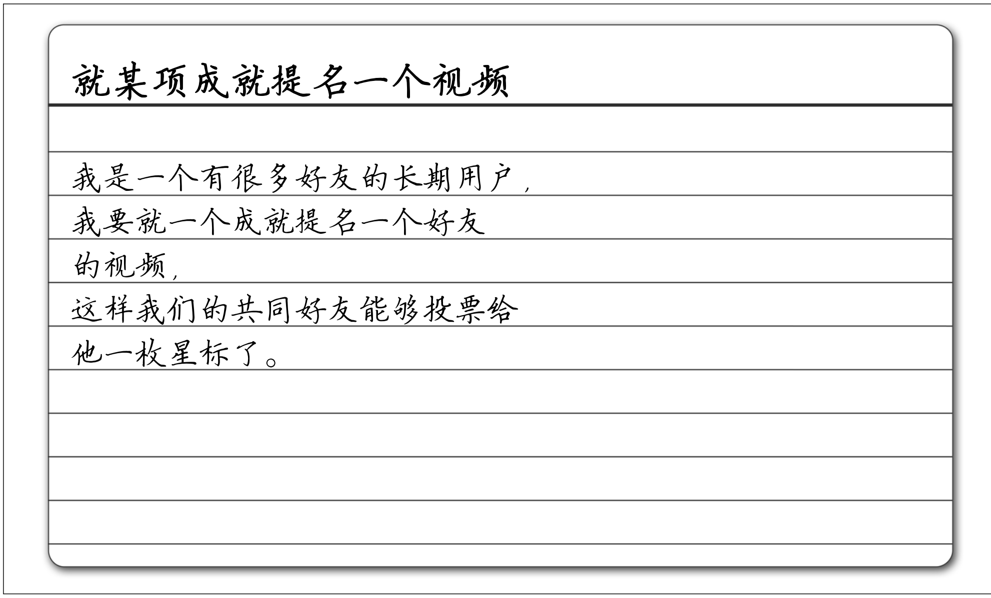


图 5-2：写在卡片上的一个用户故事

这个用户故事之所以有效，在于它明确列出了以下三件重要的事情。

- 用户是谁：一个有很多好友的长期用户
- 用户要干什么：就一个成就提名一个好友的视频
- 用户的目的是什么：为了让他们的共同好友投票给他一枚星标

这个用户故事还有一个标题“就某项成就提名一个视频”，这方便大家在讨论中提及这个用户故事。

这个用户故事的信息量其实很大。它告诉开发人员有很多东西需要开发：提名和投票的用户界面，存储提名和投票的机制，修改系统的视频显示界面来根据成就获得情况显示星标，可能还有一些其他的东西。

那些没有写入用户故事的信息也一样很重要。哪里都没有提到任何形式的规则编辑器（像他们画蛇添足那样）或者其他不必要的特性。这就是用户故事是对抗“镏金”现象的有效工具的原因。设想开发团队把用户故事落实到纸面上，与产品所有者（以及用户、利益干系人和其他有好点子的人）过一遍用户故事，然后贯彻执行下去，会发生什么？如果能够做到这些，他们开发出用户不需要也不会用到的功能的几率就会大大降低。

用户故事也是管理积压工作表的一个好工具。很多敏捷团队的积压工作表几乎就完全由用户故事构成。由于用户故事简单易懂，而且是从用户的角度出发编写的，产品所有者可以很容易地与用户和相关人员沟通，找出最有价值的功能。同时每个用户故事都很短，这样添加新的用户故事就很容易，而随时调整它们的顺序也很简单。要开始一个冲刺的时候，

产品所有者和开发团队可以从积压工作表中抽出一些用户故事，并把它们作为本次冲刺的交付目标。

然后，开发团队可以与产品所有者一起讨论用户故事的细节，确保用户故事的含义清晰准确，并确定完成标准。通常接下来一个用户故事会被分解成多个任务，然后大家对这些任务的耗时进行估计。细分后的任务会进入任务板的“待处理”一栏，等待人来处理。当一个团队成员准备开始一个新的任务时，他就选择下一个最有价值的任务，并把自己的名字写上去，然后把该任务移动到“处理中”一栏以表示自己正在处理该任务。

5.2.3 满意条件

确定你要开发什么的一个方法就是在头脑中设想它完成时是什么模样。对于一个开发人员来讲，停下来看看自己刚做的东西，然后发现它已经完成了，是一件令人感到满足的事情。“满意条件”是一个帮助开发人员知道软件完成时的样子的有效工具，同时它也有助于估计离完成还有多远。（有些人也把满意条件称为“接受标准”。）

满意条件和用户故事一样，表面上看非常简单，却达到了很复杂的目标。一般满意条件是针对每个用户故事定义的，编写完用户故事后，再把用户使用该软件能够做的一些具体的事情也写出来，形成满意条件。一般情况下，满意条件可以放在它所对应的用户故事的那张 3 英寸 ×5 英寸卡片的背面。满意条件一般由产品所有者执笔，或者至少经他们审阅。

下图所示是有关“成就”的用户故事所对应的满意条件。



图 5-3：写在用户故事卡片背面的满意条件

这些满意条件对开发人员来说很有价值，因为它们能够防止开发人员提前宣布开发完成。有时候一个开发人员完成了一个特性的很大一部分，却把最后的收尾工作留到冲刺结束时

才做，这种情况并不鲜见。例如，他可能开发完了提名页面和视频显示页面的星标，但在他把所有这些部分集成起来之前，这个功能并没有百分之百完成，趁着对这个用户故事印象深刻的时候把它一次搞定显然效率更高。

满意条件在这里起到的作用就是：给“完成”下一个具体的定义。这就给团队和产品所有者一个具体的、没有歧义的方式来确定一个用户故事完成了没有。说一个用户故事“完成了”，那么所有的理解、开发、测试和部署这个用户故事的工作都必须已经完成。只有在用户可以用该软件执行每一个满意条件（就像在冲刺评审时那样执行）时，一个用户故事才算是“完成了”。在这之前，这个用户故事都不算完成，开发人员要继续专注于该用户故事（还记得那条关于“每个人都应专注于工作”的 Scrum 价值观吗？）直到完成。一旦完成，开发人员就可以把这个用户故事从任务板的“处理中”一栏移动到“已完成”一栏。搞定！

5.2.4 故事点和速度

在做冲刺计划的时候，你的整个团队需要一起估计在本次冲刺中能够完成的工作量，以此来设置一个冲刺结束时的交付目标。可是，这个估计到底该怎么做呢？

可以这么说，有多少个团队，就有多少种估计工作量的方法。多年来，被证明对很多 Scrum 团队都有效的方法是使用“故事点”。所谓故事点，就是通过给每一个用户故事分配一定的点数，用以表示开发这个用户故事需要付出多大努力。通常，这些点数是通过将当前用户故事与过去开发过的用户故事进行比较得出的。

至于给每个用户故事分配多少故事点数，并没有一个标准，有些团队给每个用户故事分配 1 到 5 之间的故事点。（5 这个数是随意选取的，也许有人会选择 1 到 10 点，或者其他数字，只要每次冲刺都使用一致的标准即可。还有用斐波那契数列中的数字的，或者用指数数列中的数字的。你可以选择任意一种方案，只要你的团队成员觉得可以就行。）两个 3 点的用户故事应该具有大致相当的工作量。随着故事点的分配，你的团队就会开始发现他们能够在一次冲刺中完成（或者“烧掉”）相当于多少个故事点的用户故事。比方说你的团队平均一次冲刺完成大约 25 个故事点的任务量，那么他们的“项目速度”就是每次冲刺 25 点。

一般一个团队在多个冲刺中的速度是相对比较稳定的。虽然项目速度很难提前预知，但你可以通过过去的历史数据来帮你计划接下来的冲刺。

说到这，估计每个人都看到过理财产品的免责声明吧：“历史表现不代表未来收益，本公司不对投资收益提供任何保证。”这对故事点同样适用。就算你的团队上次冲刺烧掉了 32 个故事点，上上次烧掉了 29 个，这也并不能保证他们这次冲刺能够烧掉 30 个上下的故事点；在一次冲刺中，人们可能会误判某些用户故事，碰到意外的技术问题，碰巧有人休假、辞职，或者其他影响到项目进度的事件。即便如此，长远来看，故事点和项目速度对于很多 Scrum 团队都有着令人惊讶的可靠的指导意义，而你，可以在计划你的冲刺的时候，充分利用这个可靠性。

一次使用故事点方法的冲刺计划会议大致是下面这样的。

(1) 从产品积压工作表中最有价值的那些用户故事开始。
(2) 从中选择一个用户故事（最好是选最小的那个，因为它可以作为一个比较的基准），从过去的冲刺中选一个工作量相当的用户故事，将后者的故事点数赋给前者。
(3) 跟团队讨论，看看这个估计是否准确，在这个过程中如果发现额外的问题、工作量，或者技术上的挑战，则增加估计的故事点；如果有简化因素、可重用代码，或者要缩小开发范围，则减少估计的故事点。
(4) 如此循环，直到积累够一次冲刺的故事点。

别把太多的东西放进冲刺积压工作表。适当地留出一点空间可以，但是不能超出过去的量。如果团队的平均速度是每次冲刺 28 个故事点，而你们的冲刺积压工作表已经达到 26 个故事点了，那你就只能再加一个 2 点的用户故事，或者两个 1 点的用户故事。由于程序员骨子里就乐观（“我们就得这样，我们是创作者”），他们会不自主地想再加一个 3 点的用户故事，就超出 1 点而已。不要对这种诱惑让步，这容易导致冲刺评审时让用户失望。

有人可能会问，第一次应该怎么办呢？积累历史用户故事、在团队中建立类似 3 点的用户故事到底需要多少工作量这种常识是需要时间的。所以，对于第一次来说，你只能猜。选一个差不多中等工作量的任务，给它分配 3 个故事点。找到那个最难的任务，给它分配 5 个故事点。再给工作量最少的那个分配 1 个故事点。以此为基础去估计其他的用户故事，把你的冲刺积压工作表填满。经过两次冲刺以后，你就有足够的用户故事可供比较了，你也会对团队的平均速度有个更好的认识。

故事点为什么有效

与用户故事一样，故事点非常简单。上手很容易（虽然具体使用中有很多细微差别，我们这里不作详细介绍）。可是它为什么如此有效呢？

- *故事点很简单*
  故事点这种方法很好理解，很容易给新成员解释清楚。再者，故事点数通常比较少（一般几十或几百，而不是几千），相对容易计算。

- *故事点并不神秘*
  很多开发人员和项目经理都曾经历过估计不准的情况，于是他们形成一种思维定式，认为软件开发就是无法准确估计的。故事点这种方法是基于特定团队的真实的、当下的经验进行故事点分配的，它得出的估计也就没有什么神秘的了。

- *故事点可以被团队掌控*
  当老板跟你说你需要达到某种量化目标或者指标时，你会有种身不由己的感觉，而且这也会给团队带来额外的压迫感。可是，当开发团队自己决定如何度量自己的工作，而且仅把这种度量方法用于他们的计划过程，它就从外来的压力变成了一个有用的工具。

- *故事点让你的团队开始讨论工时估计*
  当一个新成员加入他的第一个 Scrum 团队，他也许是职业生涯中第一次公开谈论工时估计，甚至可能是第一次有机会在这个问题上表达个人看法。估计工时是一项技能，掌握它的唯一方法就是通过练习，而所谓练习，多数情况下也就意味着讨论某些特定任务需要多少工作量才能完成。

- *开发人员并不惧怕故事点*

  太多的程序员曾有过这样的经历：给项目经理一个未经深思熟虑的大致估计，结果项目经理就把这个不成熟的估计作为项目的铁打的截止时间。采用故事点的方法就不会发生这样的情况，因为故事点绝不会转换成小时数或者日期，只有一个日期是确定的，那就是冲刺的结束日期，而许诺要在截止时间前完成的工作可以在每日例会中进行调整，这是整个“可见 – 检查 – 调整”周期的一部分。

- *故事点可以帮助团队认识到一个用户故事的确切含义*

  如果对于同一个用户故事，你认为应该给它 5 个故事点，而我认为应该只给它 2 个，那么我们可能并不是在工作量上有分歧，而是在对该用户故事的具体含义的理解上有分歧。如果我们能够把这个问题说透[3]，也许我们最终发现，我以为我们需要开发一个简单的命令行工具，而你以为它需要一个图形界面。在计划阶段发现这种分歧是一件好事，这好过开发一半的时候才发现一个 2 点的用户故事其实应该是 5 点。

- *故事点可以帮助团队中的每个人投入进来*

  故事点和速度给了每个人一个大家都认同的客观的参考系：我们上次冲刺烧掉了 26 个故事点；我们都同意这个用户故事差不多值 4 个故事点。如果你是团队中的鸡，你会倾向于不参与这些事，最后你会发现大家没有你也一样向前推进了。你发现你所了解的信息可以帮助团队给一个用户故事分配 3 个故事点而不是 1 个的那一天，就是你开始关心冲刺计划并保持工时估计现实、合理的一天，这就是你变得由衷地投入的过程。

5.2.5 燃尽图

*燃尽图*是一种让任何人都可以一眼看出当前的冲刺与团队过去的速度相比如何（是更快还是更慢）的方法。下面是基于故事点画出燃尽图的方法（这个方法同样适用于其他度量单位，比如小时数，但我们这里使用故事点数）。

(1) 以一个空白的折线图开始。*X* 轴是日期，从冲刺的第一天开始，到最后一天为止。*Y* 轴是故事点数，从 0 到本次冲刺积压工作表中总故事点数的 1.2 倍。在图中画下第一个点（第一天，本次冲刺总故事点数），将这个点与项目结束点（最后一天，0 个故事点）用一条直线连接起来，就得到了所谓的“指导线”。

注 3：一起面对面地沟通来给另外一个用户故事打分，而不是通过全面的手册说明。

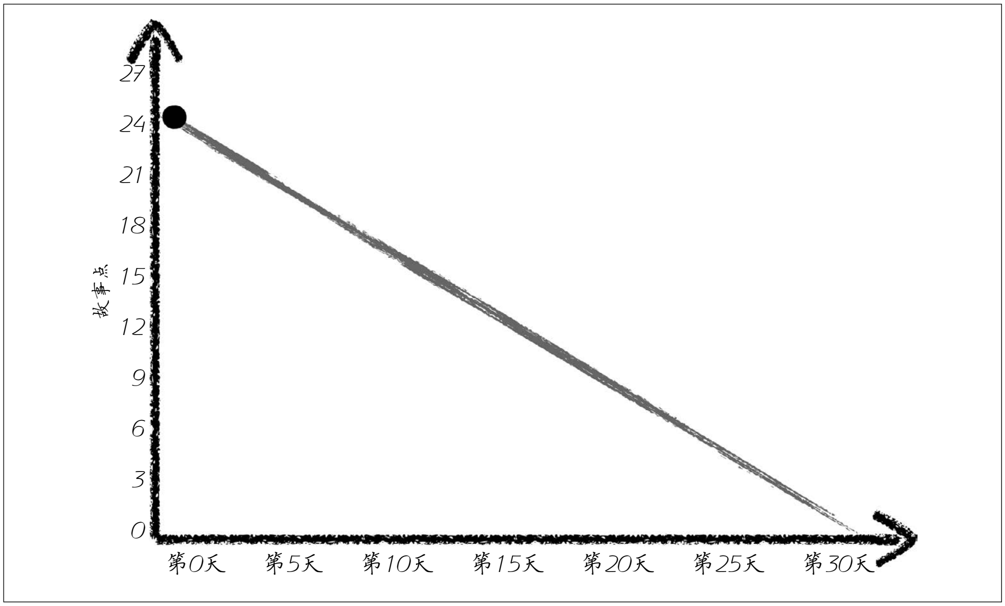


图 5-4：冲刺刚开始时的燃尽图。我们这里使用故事点创建了这张图，但你也可以使用小时数、天数或其他度量单位

(2) 一旦第一个用户故事完成并移动到任务板的“已完成”一栏，就在图中画下下一个点（当前日期，本次冲刺剩余的故事点数）。随着更多用户故事的完成（从积压工作表中烧掉更多的故事点），逐步往燃尽图中添加更多的点。

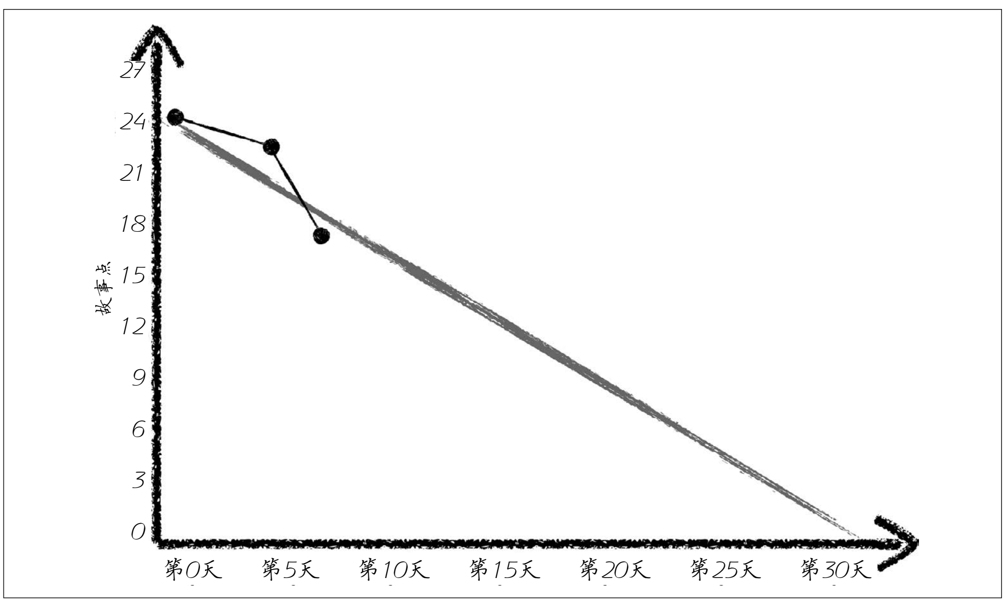


图 5-5：两个一共 7 个故事点的用户故事被烧掉了

(3) 你可能在每日例会中发现需要增加新的工作。有时候你这么做的原因是开发团队的进度比预期的要快，于是他们能提早完成。或者，来了一个重要的支持任务，开发团队和产品所有者都同意这个任务应该添加到当前冲刺中，但是他们并不知道应该相应地移除多少工作来让当前冲刺的总工作量保持不变。当你把新的任务卡添加到任务板上时，约定一个跟进会议，让整个团队一起对新增任务进行工时估计，并把它们加入燃尽图。在新增的点旁画上额外的线作为额外的标注会有帮助的，而且你大可以在燃尽图上做些笔记，没事的！

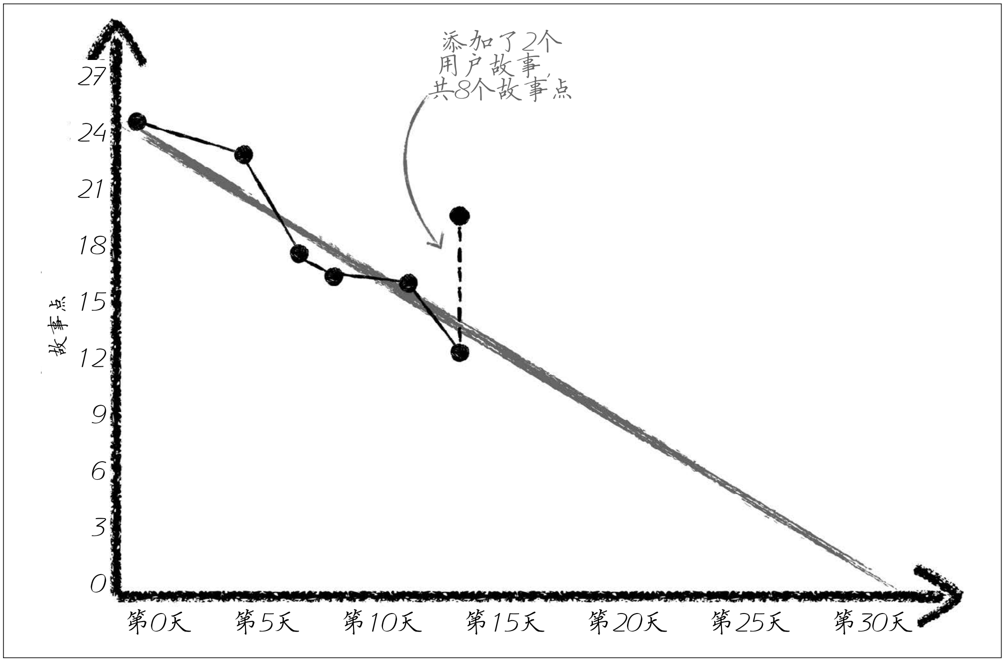


图 5-6：产品所有者在冲刺半途加入了新的用户故事

(4) 随着你接近冲刺的尾声，越来越多的故事点被烧掉。留意你的真实进度与指导线之间的差距，如果你的真实进度曲线总是高于指导线一大截，那就说明你的冲刺中的故事点过多，需要适当移除一些用户故事了。

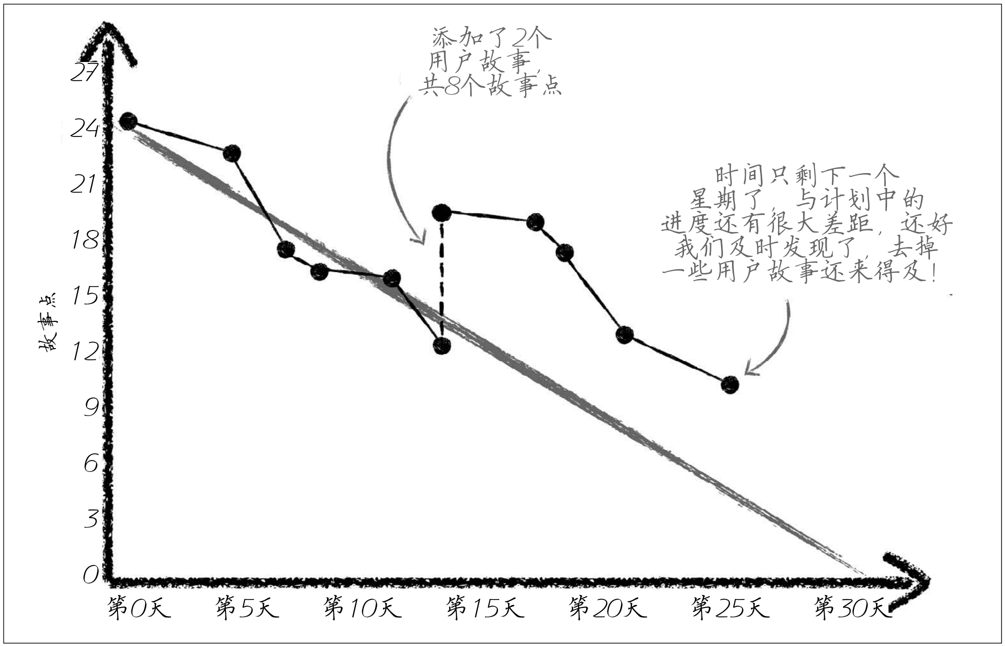


图 5-7：实际进度与指导线之间的差距表明你很可能无法在冲刺结束前完成所有用户故事

有很多软件可以帮助你管理积压工作表、用户故事以及故事点，并且帮助你自动画出燃尽图。不过，很多团队更喜欢自己手动画燃尽图，并把它与任务板挂在同一面墙上（其实二者经常被画在同一个白板上）。这样大家可以随时看到项目的进展情况。如果每个开发人员都能有机会完成一个用户故事、把它移动到“已完成”一栏，并更新燃尽图，那将会给整个团队带来很强的满足感。

5.2.6　通过用户故事、故事点、任务和任务板来计划并实施冲刺

如果你读过 Scrum 指南（或本书第 4 章开头部分）中关于如何计划一次冲刺的介绍，会发现做冲刺计划就是要回答以下两个问题。

- 本次冲刺需要交付什么？
- 整个团队要如何完成这些工作？

我们刚刚介绍了如何使用故事点和速度这两个工具来决定把哪些功能包含进一次冲刺。这种方法是 Scrum 团队在冲刺计划的前半程常用的方法。可是，冲刺计划的后半程呢，如何安排所有的具体工作呢？

对此，最常见的做法是，为每一个开发任务都建立一个任务卡。这些开发任务可以是任何需要完成的事情，如：编写代码，系统设计和架构，构建测试，安装操作系统，设计并搭建数据库，将系统部署到生产服务器，进行可用性测试，还有所有你能想到的一个团队进行软件开发和发布所需要做的事情。

下面是常见的做法。

(1) 召开第二次冲刺计划会议。Scrum 主管来主持，从第一个用户故事开始，逐个讨论每个用户故事需要做哪些具体的工作才能实现。大家一起得出一个任务列表，列表中的每一个任务都是大家认为可以在一天之内完成的。每个任务都要写在一张单独的卡片上（有人把用户故事和开发任务写在不同颜色的卡片上以便区分），把故事卡和与它对应的任务卡放在一起。如此往复直至所有用户故事都计划完毕。如果在规定的冲刺计划时限内没能讨论完所有用户故事，那么每一个未处理的用户故事就会得到一张任务卡，任务内容就是讨论该用户故事的计划。

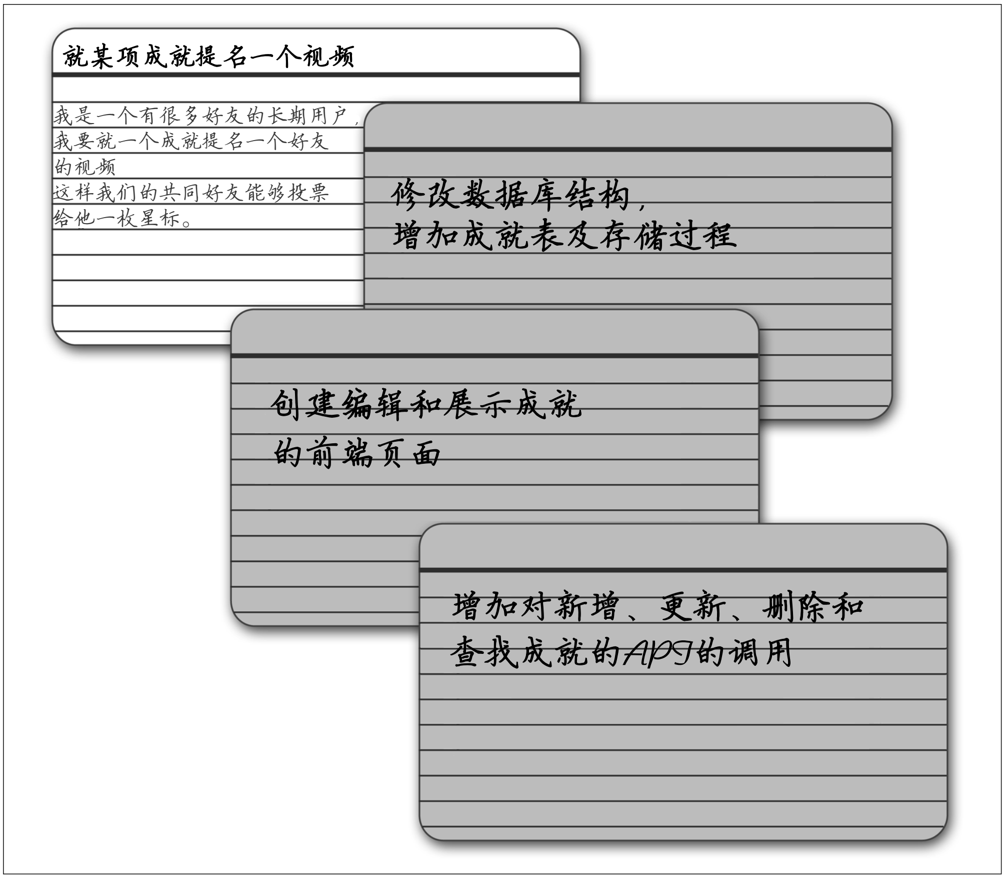


图 5-8：冲刺计划的后半程就是把用户故事拆散成具体可执行的任务

(2) 用户故事卡及其对应的任务卡被放在一起，并添加到任务板的“待处理”一栏。

(3) 团队成员完成了一个任务并准备开始下一个任务时，他就把已完成任务的任务卡移动到“已完成”一栏，然后，把接下来要处理的任务从“待处理”一栏中移出，写上他的名字，再把它放回到“处理中”一栏。如果该任务所对应的故事卡还在“待处理”一栏，那就把该故事卡也移至“处理中”一栏（但是他不应在故事卡上署名，因为团队中可能有其他人也在处理该故事所对应的其他任务）。

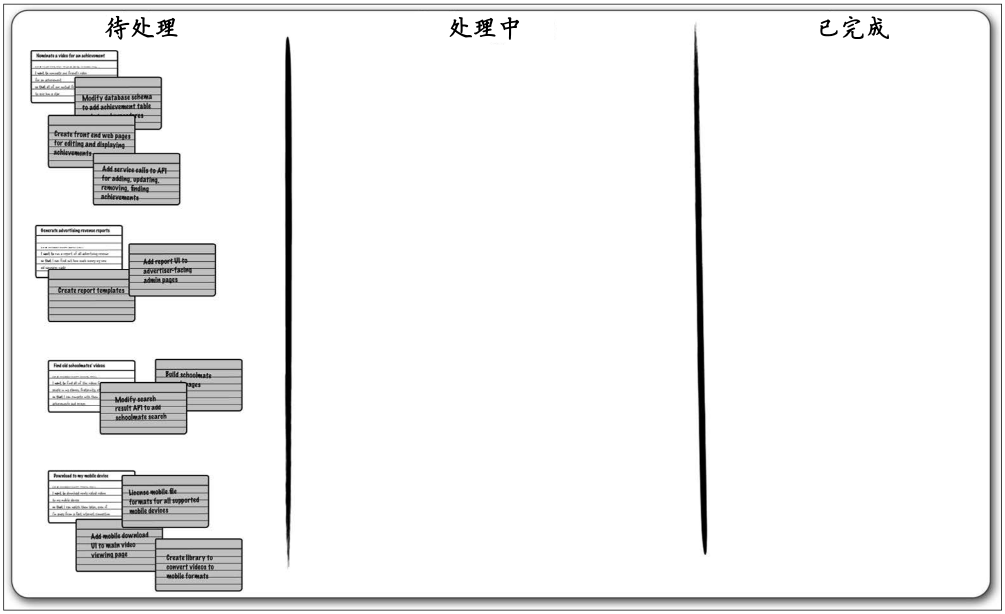


图 5-9：每个故事卡都跟它的任务卡放在一起，贴在任务板的“待处理”一栏中

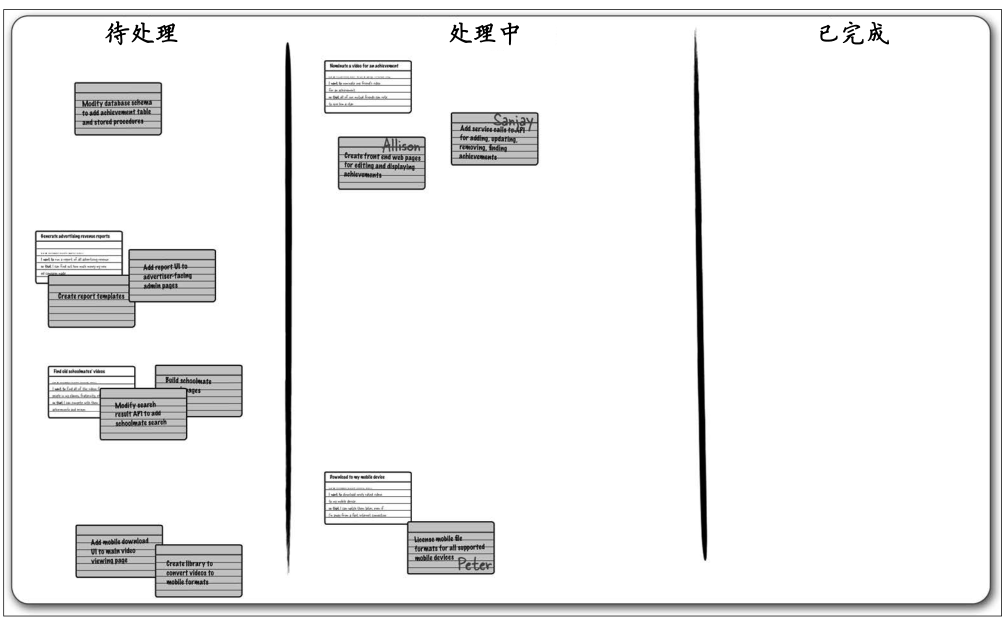


图 5-10：每个团队成员一次只处理一个任务：先在任务卡上署名，再把该任务卡移动到任务板的“处理中”一栏

(4) 随着冲刺的推进，任务卡逐渐从“待处理”被移动到“处理中”，再到“已完成”。在这个过程中，发现某个用户故事需要做一些额外的工作才能完成是常有的事。如果发生这种情况，就要把新增的任务添加到任务板，而做这件事的人需要在每日例会上通知大家，以便每个人都知道发生了什么并能够帮助发现潜在的问题。

(5) 一旦一个团队成员完成了某用户故事的最后一个任务，他就把对应的故事卡从任务板上揭下来，确认所有的满意条件都已达到，然后把该故事卡移至“已完成”一栏，与它的任务卡放在一起。（注意：在一个用户故事得到产品所有者代表公司认可它为“可以发布”之前，不能算作“完成”！）如果他发现某项满意条件没有达到，这就说明还有一些工作没有做，所以，他应当把该故事卡再放回“处理中”一栏，并在“待处理”一栏中为完成该用户故事所需的那些工作增加新的任务卡。

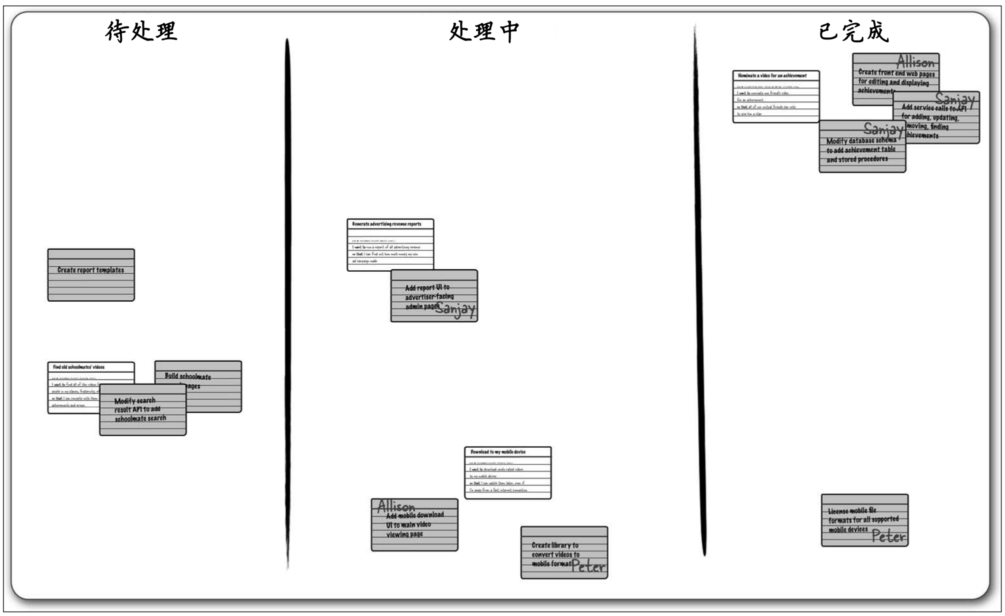


图 5-11：当某团队成员完成一个任务时，将任务卡移动到“已完成”一栏，并申领一个新任务，如果对应故事卡的所有满意条件都已达到，就把该故事卡也移动到“已完成”一栏

(6) 一旦截止时间来临，冲刺就结束了。这意味着也许会有一些用户故事和任务还在“待处理”和“处理中”这两栏里。这些没来得及处理的用户故事应该放回到产品积压工作表中，等待其优先级在下一次的冲刺计划会议中再次被评估 [4]，而它的故事点数则可以根据剩余工作量的多少适当下调。只有那些被移动到“已完成”一栏的故事和任务才算数，其他的不算。

注 4：关于是否应当把未完成的工作放回到产品积压工作表中并重新评估其故事点数，Scrum 培训师之间是有一些争议的。有些人认为这没什么问题，也有人会告诉你不要更改分配的故事点，因为这样更有利于团队估计。还有人认为重新评估与否并不重要，因为最终这些都会被遗忘。

5.2.7 广受认可的Scrum实践

如果你翻阅 Ken Schwaber 关于 Scrum 的原版书《Scrum 敏捷项目管理》，你会发现书中并未提到用户故事和故事点。大家都是从其他地方学到这两个技巧的，比如 Mike Cohn 的大作《用户故事与敏捷方法》[5]。

有很多能够帮助改善 Scrum 的实践，这当然不意外。我们之所以要开回顾会议，就是为了寻找更优的做事方法。如果我们把某个人在某一本书中所写的内容当作金科玉律而不敢越雷池半步，所谓的“追求卓越”不就失去意义了吗？

也正因此，大家普遍都会采用额外的工具和技巧，Cohn 把这些工具和技巧称为广为认可的 Scrum 实践（Generally Accepted Scrum Practices，GASP）[6]。例如，很多团队发现他们站着开每日例会（与会人员全程站立）时效率更高。站着并非 Scrum 最原始的形式，却被众多 Scrum 团队广泛采用。

回忆一下敏捷宣言中的最后一个原则：

> 团队定期反思如何提升效率，并依此调整自己的行为。

在回顾会议中需要牢记这一原则。要尝试寻找改进的新办法，但是不要放着现成的方法不用。如果你在实践 Scrum 的过程中遇到了问题，很可能别人早就遇到过了。如果有个人从旁指导（就像 Roger 和 Avi 有 Eric 指导一样），就能帮你指出化解问题的方法。

要点回顾

- 用户故事之所以能帮助开发团队理解用户，是因为它清晰地定义了用户需求以及软件应当如何满足该需求。
- 每个用户故事都有一组**满意条件**来帮助开发人员知道该用户故事怎么才算完成了。
- **故事点**和**速度**用来估计一次冲刺能够容纳多少个用户故事。
- 把**燃尽图**张贴在一个显眼的地方，让所有人看见，能够帮助大家了解都有哪些任务完成了，哪些还没有完成，以及项目进展是否顺利。

故事：在一家小公司中有一个负责开发一款手机应用的团队

- Roger——尝试采用敏捷的团队主管
- Avi——产品所有者
- Eric——另一个团队的 Scrum 主管

注 5：《用户故事与敏捷方法》，Mike Cohn 著。

注 6：出自论文“GASPing About the Product Backlog”（http://www.mountaingoatsoftware.com/blog/gasping-about-the-product-backlog），2014 年 7 月 26 日收录。

5.3 第6幕：第一次胜利

已经半夜 11:30 了，所有人都还在办公室，但并非为了加班。那个星期早些时候，他们正式发布了 Lolleaderz.com 并受到了很高的赞誉，也收获了众多的用户。CEO 听说了 Roger 和 Avi 在计划一个小型的庆功会，但他觉得那么搞太寒酸了。结果，CEO 包了酒席，请来了 DJ，还给大家提供了一个开放式吧台。现在，Roger 和 Avi 都已经酒至半酣了。

Roger 晃晃悠悠来到 Eric 面前，后者正在跟其他几个团队成员闲聊。Roger 说："没有你就没有今天的庆功宴啊！"

Eric 想了一下说："谢谢你的恭维。不过你回头想想我帮了什么忙。我没有帮你们做任何实际的工作，对吧？我不过是指出了一些问题，并告诉你我的解决方法，再有就是偶尔让你真正遇到一些问题，并且自己去发现问题到底出在哪。"

Avi 也走过来，说："哎，那几回怎么说？有几次我们做得不是很顺。就不能避免类似的情况吗？我这绝不是抱怨啊，结果已经说明一切了。"

这时，团队中的大多数人还有几个销售经理已经聚过来了。Eric 说："如果没经过那次跟销售经理一起开的尴尬的会议，你们会想起来使用用户故事吗？你们能学会用项目速度来计划冲刺吗？"

Roger 思索了一下，说："你等会儿。你的意思是不是说：不撞南墙不回头？"

Eric 说："应该说：吃一堑，长一智。正是你们遇到的问题给了你们一个理由，齐心协力去寻找更有效的工作方式。找到了，整个团队就一起成长了。"

大家都纷纷点头表示同意，屋子里响起了"嗯""说得对"之类的赞同声音。最后，后面传来一个声音："这才是我想听到的！"

是 CEO 在说话。他满脸笑意："我看好你们！干得漂亮！"

5.4 回顾Scrum价值观

在 Lolleaderz.com 项目中 Roger、Avi 以及整个团队遇到了很多问题。对很多团队来讲，这些问题会毁掉整个项目，并导致像 CHAOS 报告中所描述的那种失败。这个团队是如何把危机变为机遇的呢？他们是如何从中汲取经验并最终获得成功的呢？

一个重要的因素在于 Roger 和 Avi 有秘密武器：一个给力的导师，Eric，帮助他们绕过礁石险滩。Eric 所做的第一件事情就是帮助 Roger、Avi 及团队的其他成员理解 Scrum 的价值观：承诺、尊重、专注、开放和勇气。他用"猪"和"鸡"的比喻来解释所谓"承诺"的真正含义；他帮助团队理解应该如何开每日例会，以此来保证他们能够保持专注。另外，他还向所有人展示了如何通过倾听彼此来相互尊重。通过使用用户故事和速度的概念来改进冲刺和积压工作表的计划与评审，Eric 让团队中的每个人（甚至包括那些销售经理和 CEO 本人）了解项目的进程，从而在整个团队中建立了一种开放的态度。而当面对困难的时候，Eric 让大家意识到他们有勇气面对事实，即使这在短期内可能会让一部分人不高兴。

每个项目都有其问题与挑战。尤其在你首次尝试某种敏捷方法时，这些问题会被缺乏经验、理解偏差以及常犯旧错这些因素所放大。像 Eric 这样的优秀导师对此十分了解。他并没有尝试阻止这些错误的发生，相反，他巧妙利用这些错误为整个团队制造了学习的机会。这种做法被称为“允许失败”，而这也正是教学的一个基本原则。

Lyssa Adkins 在《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》一书这样写道。

> 要允许失败：当然，这并不是说在团队即将冲下悬崖的时候你应该袖手旁观，而是要利用每次冲刺中的各种机会让他们适当地失败。相比于把他们保护起来，让他们共同经历失败并从失败中找到出路可以更好地锻炼他们。你以为会给他们带来不利影响的那些东西，也许恰恰对他们最有帮助。不信走着瞧。[7]

看到勇气、开放、承诺、专注和尊重这些字眼并表示万分认同，这是一回事；走进老板的办公室并告诉他由于本次冲刺时间有限，团队无法完成某项特性，这就是另外一回事了。如果你面临丢掉工作的风险，鼓起勇气是很困难的。

不过，如果不能坚持这些价值观，你就无法成功地实施 Scrum。可惜的是，尽管很多公司的文化与 Scrum 的价值是兼容的，但也确实有一些公司不是这样。甚至你的老板要的是应声虫式的属下，如果你尝试创建一个自发组织的团队，他就会炒你的鱿鱼，这也完全是可能的。

Scrum 团队会从他们的错误中总结经验，这正是他们成长的方式。团队中需要一种允许犯错的文化。因此，Scrum 团队更强调协作，而相互尊重则是重要的 Scrum 价值观。可是，就算一个团队能够容许犯错，也能从错误中学习，这个团队所处的公司却不一定能够接受错误。下面是创新软件工程师 Grady Booch 在他的《团队之美》一书中的一段表述。

> 一个组织是否健康的一个标识就是他们是否对失败讳莫如深。那些完全不能容忍失败的组织往往是最没有创新的，而且他们也全然无趣，因为这些人惧怕失败到了只会采取最为保守的措施的地步。另一方面，那些把失败看得很淡的团队（还没到允许完全毁掉整个企业的那种失败的地步，但是允许一定程度的失败），往往更加高产，因为他们不会在每一行代码上都谨小慎微。

那么，如果你所在的就是那种一旦失败就可能被炒鱿鱼的公司，该怎么办呢？这种情况下还有可能使用 Scrum 吗？

5.4.1　具体实践没有价值观也有效果（只是别管它叫Scrum）

并不是每个公司的每个团队都能在一开始就实现自我管理，这很正常。如果你的公司文化与 Scrum 的价值观不吻合，那么作为一个 Scrum 信徒，你的任务就是帮助周围的人理解这些价值观。最好的方法就是以身作则：向身边的人展示什么是对工作保持开放的态度，尊重团队中的其他成员，并在面临问题时展现你的勇气。找个靠谱的导师能够让大家意识到这种方法真的有前途，并帮助大家改变态度。开发团队经常找一个顾问做导师，因为当改变做事方法的建议来自外部时，公司的老板更容易接受。

注 7：《如何构建敏捷项目管理团队：ScrumMaster、敏捷教练与项目经理的实用指南》，Lyssa Adkins 著。

但有时候即便你做对了所有的事情，公司的主流文化可能还是跟 Scrum 的价值观搭不上边。如果是这样的话，自我组织和集体承诺也许没法应用到你的团队中。正如 Ken Schwaber 告诉我的，如果不能达到自我组织和集体承诺，那也就不叫 Scrum 了。

但这也不是什么问题呀！

就算没有所谓的 Scrum 价值观的“灵魂”，就算没有自我组织和集体承诺，Scrum 还是能够提供一些非常不错的实践方法。这些方法简单易行，而且一般说来不需要事先经上级批准（也不需要事后道歉）即可在你的团队中实施。

如果你所处的环境无法完全采用 Scrum 价值观但允许你采用部分实践方法的话，你还是能够得到比完全不采用这些实践方法更好的效果。既然如此，何乐而不为呢？只要确保你的团队知道自己在干什么，并把握好行为的边界即可。

但是，不要把采用 Scrum 的实践方法等同于完全采用 Scrum。如果你这样做了，只会以后制造麻烦。那些听说过 Scrum 能打造“超级高产团队”和带来“惊人成绩”的同事会质疑：为什么你做不到这些？而你没法给出令人信服的答案。更糟的是，你的团队很可能为 Scrum 感到热血沸腾，最后却发现生活不过是改变了那么一丁点儿而已。某种程度上讲，他们可能会觉得他们也就能做到这样了，而如果“这样”跟之前没有多大区别，那是相当让人沮丧和消极的。这些问题会给将来的改进尝试带来负面效果：因为把标准降低了，他们会认为他们所面对的限制是无法掌控的。

换句话说，如果你跟所有人说：“恭喜！你们也在使用 Scrum 了。”这将给人一种印象：Scrum 不过是把进度报告会改成了每日例会，把需求变成了积压工作表而已。这样做是危险的，因为最初你通过实践 Scrum 所尝试解决的基本问题依然存在，所以大家会认为 Scrum 无法解决这些问题，更糟糕的是，他们可能会认为这些问题到哪里都会存在，不是人力所能够解决的。回想一下你有多少次听到有人说“开发人员无法做估计”，或者“软件就是 bug 多多”，或者“软件项目没有不延期的”，就好像这些都是天经地义、不可改变的。

最糟糕的是，当你不停地声称 Scrum 能够解决某些显然没有被解决、大家也不认为能解决的问题时，你就变成了固执的敏捷狂热分子。人们会因看不到效果而对宣扬 Scrum 的声音产生反感。这对你的职业生涯不利，而且会让你周围的人对 Scrum 失去兴趣。

你应该换一种方式！

不仅要让团队采用一些马上就可以用的方法，还要让他们看到将来他们能够走得更远。这样一来，他们就不会因为知道了些皮毛就以为自己已经看到 Scrum 的全貌了。

设想一下，如果你公开表示你们不过是采用了 Scrum 的一些实践方法，但还没开始实行自我管理和集体承诺，会怎么样？与其宣称这种“有实践没灵魂”的 Scrum 是完整的，倒不如坦诚地告诉大家我们只是采用了一些 Scrum 实践，但距离 Scrum 的价值观、自我组织、承诺并创造价值，还有很长的路要走。

在这种坦诚的表述中，你并未作出你无法实现的承诺。你告诉大家你会让事情有一点改善，这样，当他们看到这样做比什么都不做会强一些时，就不会因为你没能解决公司所有的问题而失望，相反，他们会因整个团队能够生产出更好的软件而且开发得更快更轻松而

感到高兴。当你有了一次次的成功（即便是很小的成功）作为后盾时，再跟大家讲公司价值观层面的东西就容易多了。而从老板的角度看，你不是批评他不够开明，而是眼下就拿下比以前更好的业绩，并承诺将来给他更好的业绩（如果他能让你的团队就那些新价值观做些独立尝试的话）。

5.4.2 你的公司文化与Scrum的价值观兼容吗

不是每个公司都准备好了放弃上令下行的项目管理模式并转向自我管理，也不是每个团队都能理解集体承诺的精髓。你的团队或公司是否适合采用 Scrum，要看团队或公司的文化与 Scrum 的价值观是否兼容。可是，Scrum 价值观要如何对应到现实生活中的做事方法呢？

你可以问自己以下这些问题，并给出诚实的回答。与你的团队、经理讨论这些问题。如果每个人都确实认同这些东西，那么你的公司或团队就可以开始采用 Scrum 了。如果不是这样，你们的讨论也应该对如何把 Scrum 引入你的团队提供了一些有帮助的信息。

要确定你们是否能做到*承诺*，问一下你自己、你的团队，还有你的老板是否对以下这些东西表示认同。

- 放弃对项目的控制，信任开发团队，并让他们决定应该交付哪些功能。
- 绝不在没有征求团队里其他人意见的情况下开发某个功能并最后把它集成到整个系统中。
- 没有单一的责任人（替罪羊）。[8]
- 倾听他人的评论和反馈，而不是让别人管好自己的事情。
- 心甘情愿地负起责任，团队中的每个人都应如此。

要确定你们是否能做到*尊重*，问一下你自己、你的团队，还有你的老板是否对以下这些东西表示认同。

- 充分信任开发团队，相信成员会作出正确的决策并基于项目进展和功能特性的相对价值在最适当的时间交付每一个功能，即使这可能导致项目延期。
- 给开发团队充足的工作时间，不要求他们加班加点。
- 允许开发团队根据需要自己选择任务，而不是给他们分配角色。
- 没有理由说“我也不知道他们为什么这么搞，我没参与当初的决策”。

要确定你们是否能做到*专注*，问一下你自己、你的团队，还有你的老板是否对以下这些东西表示认同。

- 不让团队中的任何人去做不在当前冲刺计划中的工作。
- 不让团队中的任何人去做未经全体成员同意的工作。
- 把对公司最有价值的工作放在第一位。
- 不能事先要求团队成员以某一特定顺序完成某些具体的任务。

要确定你们是否能做到*开放*，问一下你自己、你的团队，还有你的老板是否对以下这些东西表示认同。

注 8：要慎用“替罪羊”这个称呼，因为在很多公司，这是产品所有者的昵称，可能团队内部能够理解并做到自我管理和集体承诺，但如果公司的其他人不能，那么对这些人来说，产品所有者就是那个“替罪羊”，项目出了什么岔子都要他来负责。

- 倾听他人的意见，而且真正把这些意见当回事。
- 如果你以前从未考虑过项目规划或者从未从用户的角度考虑过问题的话，现在必须得考虑了。
- 如果你以前从未考虑过技术细节的话，现在必须得考虑了。
- 你还得考虑坐在你旁边的程序员在干什么，并且要考虑他做的事情是否与总的目标相吻合。
- 你旁边的程序员也要关心这些事。

要确定你们是否能做到*勇气*，问一下你自己、你的团队，还有你的老板是否对以下这些东西表示认同。

- 问题不能单纯归咎为项目经理计划不足。
- 不能把糟糕的需求分析归咎于产品所有者或经理。
- 要用心理解你的用户。
- 容许不完美，因为用户最需要的是"够用"。

如果对上述所列事项的反应都是"可以"，那么你的团队、经理和公司很可能具备与 Scrum 价值观相匹配的文化。相反，如果无法接受上述事项中的一项或几项，那么你应该与团队和老板进行一次坦诚的讨论。如果你感觉这种讨论不太可能的话，那么要想充分发挥 Scrum 的作用，可能得在"开放性"上下下工夫了。

常见问题

积压工作表和任务板不就是项目日程表吗？ Scrum 团队的做法真的跟我的团队的行事方式有什么不一样吗？

以传统的方式使用 Scrum 中的工具和实践方法是完全有可能的。比如有些团队会把用户故事写得很像传统的用例。可是如果一个团队不改变他们的做事方法，也就没法得到更好的结果，这样的团队并未对他们的思维方式进行必要的调整来适应 Scrum 的要求，而结果则是他们的生产效率并未得到太大的提升（当然，比完全不做改变要强一点）。

高效的 Scrum 团队与传统项目团队的一个关键区别在于：他们的计划不会被束之高阁，仅仅在进展不顺利时才拿出来修改。他们在每天的例会上都会检查计划中的问题。不仅如此，很多 Scrum 团队每周至少会拿出一小时与产品所有者一起来更新积压工作表（有人把这称为"梳理"）。产品所有者总是会不断想出可能有价值的新功能。在每周梳理积压工作表的过程中，开发团队与产品所有者一起创建新的用户故事卡及其对应的满意条件。他们会进行故事点估计并给新的用户故事分配故事点，然后与产品所有者一起根据价值大小评估新用户故事的优先级。

这给开发团队提供了两个非常有力的工具。一个是他们有更多的机会练习估计工作量，这样，当需求变动时，他们就清楚地知道如何分析其影响（直接进入"估计模式"，就像他们每周都做的那样）。

另外一个工具是“时刻与产品所有者沟通哪些用户故事最有价值”的能力。如果开发团队要使这种沟通更有效，必须真正理解项目的目标。这就是强调目标的重要性所在：如果产品所有者能够让团队清楚地知道为什么他们的用户需要这个软件，那么找出产品积压工作表中最有价值的功能就容易多了。

这些做法之所以有效果，是因为团队已经习惯于时刻与产品所有者一起工作并理解积压工作表中每个用户故事的价值所在，是因为团队成员不停地在计划、估计这些用户故事。这样一来，每当需求变化时，团队中的每个人都可以切换到估计模式，就像他们每周都做的那样。而这就是在最后责任时刻做计划的真正含义：团队已习惯于随时做计划，并留出特定的时间来做计划，因此这些对他们来讲都习以为常。

可是程序员不是不善于做计划吗，尤其是软件项目，从根本上就是没法预知的？

这也许是对计划最大的误解，或者换个更糟糕的说法：自以为是地想当然。这并不是说程序员善于计划。事实是没有人擅长做计划，起码一开始是这样。如果人能够一直计划周全，那做买卖就没有不成功的了，市场大环境就一直都是牛市了，每一对夫妻就都能白头到老了。（哪有人会在婚礼上计划离婚的事情呢？）

做计划并不等于预测未来。很多企业里都有人相信所谓“完美项目计划”的神话，即开发人员可以给出分毫不差的估计，而项目经理能够预料到所有风险和偶发事件。这些人的计划变动的频率跟别人没什么两样，也许他们的计划变动更频繁，因为他们常常在初期把过多的细节写进计划里，而细节出错的几率更大。难怪他们的项目总是出问题，并因此认为程序员不善于做计划，软件项目是不可预测的！

Scrum 团队的计划更有效是因为他们的计划是自上而下的。他们首先勾勒出产品的大框，从一个精简的产品积压工作表开始。这个积压工作表中只包含那些足以让产品所有者和用户判定功能价值高低的信息。在每次冲刺的开始阶段，开发团队才着手做更详细的计划，且只针对那些他们打算包含在当次冲刺中的积压工作表条目。绝大部分针对每天、每小时的计划留待每日例会上处理。如果这类细节计划确实需要在冲刺一开始就做，他们也不会刻意地不做，不过这种情况很少见，所以也不会占用过多时间，而且整个团队能够在每日例会上对这些计划进行回顾和调整。

比起后面不停地回顾和调整，一次性提前计划完毕不是更高效吗？

说实在的，不见得。很多团队在实践中发现，把决定留到最后责任时刻才做会有更高的灵活性，而且这有助于他们更高效地计划（或者决定不做计划）。这种计划方式效果好的原因在于，它把决定权留给那些最了解情况的人，留到问题被充分理解的一刻。

很多大型计划失败的原因在于：在项目一开始就制订细致入微的计划是不现实的，开发人员根本不具备制订这种计划所需要的信息。这会导致大家都熟悉的互相推诿：项目经理埋怨开发人员估计不准，而开发人员则埋怨项目经理的计划不灵。确实有人遭到埋怨，但最后没人需要承担后果，或者每个人都得承担后果，谁也不能例外。不管是哪种情况，真正受到伤害的是软件的质量。

Scrum 能使我们免于进入这种互相推诿的境地。方法就是仅基于开发人员现在所了解的信息来做计划，并允许把尚不明了的细节留待最后责任时刻再处理。我们每天都评估、

检查并调整计划，这样当计划有错误（这是一定的）的时候，我们就能够快速反应。反过来，Scrum 要求我们由衷地投入到工作中，构建并交付我们所能开发出的最有价值的软件。

每次冲刺结束的时候都能拿出可以做现场演示的软件，这现实吗？如果做出来的东西没法演示怎么办？

这是一个很好的问题，很多刚开始采用 Scrum 的团队都会问到。有些功能（像新的网页应用的页面、新增的按钮，或者终端用户行为的改变）很容易演示，只需拿出用户故事卡，启动软件，然后把满意条件逐个过一遍就行了。但有些功能很难演示。如果开发团队花了几周的时间来优化数据库、修改后台服务，或者做其他非功能性修改，你应该怎么做呢？

每一个软件更改都可以找到演示的方法，寻找合适的演示方法其实对程序员是很有帮助的。比方说某个冲刺是专门针对数据库修改的。开发人员不会说进入数据库，做了修改，然后不做任何的确认和检查工作就宣告万事大吉的，没有谁是这么编程序的！你会写代码来做一些插入、更新、删除或者随便什么其他需要做的事情来确认数据库的变更确实生效了，而且你很可能还要修改系统的其他部分来使用新的数据库。

这时就能看出冲刺评审会议上给产品所有者、用户和利益干系人的演示有什么帮助了。对开发人员来说，写一些临时代码来测试数据库能正常工作是常有的事，可是如果他们知道自己需要在冲刺结束时对此做演示的话，就会利用这个机会把那些写完就扔的临时代码保存起来，做成一个小程序（可能是一个小的控制台程序，或者可能只是提交到代码库中的几个小脚本），并花一点额外的时间打磨它，以便能够进行演示。这也许花不了太多的额外时间，但是现在他们有了一种测试变动的方法，如果后面他们需要对数据库做其他的改动，这个演示程序就能派上用场了，现在他们有了一些脚本可以重用或者作为基础进行开发了。

这种方案一样适用于其他类型的非功能性修改（即不会改变用户使用软件的方式，但是代码会发生变动的修改）。性能上的修改可以通过修改前后的性能测试来展示性能的提升。服务架构上的修改可以通过一个简单的从新的服务 API 接收各种不同数据的小程序来进行演示。总有一种方法可以演示软件的变动，做演示与不做演示相比只会给团队带来好处。这也是 Scrum 帮助团队进行计划和改进的一个方式。

在演示非功能性修改的时候，一般就是执行几个脚本、测试、程序，或者其他测试该修改的代码。开发团队要注意使用用户能理解的语言，不过，你也会惊讶地发现，用户的理解能力经常超出开发人员的预期。最重要的是，这种演示让用户和利益干系人能够亲眼看到开发团队的工作成果。没有这些演示，在不了解情况的人看来，开发团队浪费了一次冲刺，什么都没做出来。而这些演示则帮助开发团队向所有人说明了为什么软件开发要花这么多时间，这对于将来团队设定现实的截止时间会有帮助，而且还能避免不懂软件的老板给团队施压，要求提前那些截止时间。

当生产环境中的软件出现 bug 时，我们团队必须停下手头的工作去解决，所以没法等到冲刺结束再说。对支持类的任务，Scrum 是不是不太适合呢？

每个软件开发团队都要随时面对突发的状况。软件支持是一个很好的例子。一个网站开

发团队可能会遇到一个必须马上发布的补丁，而且可能还得手忙脚乱地发布。如果补丁没法等到冲刺结束再发布，那你就只有一个选择了，那就是现在就搞定它。

一个 Scrum 团队其实比传统的上令下行的、需求先行（提前起草大量详尽的需求文档）的团队更善于处理这类问题。原因是 Scrum 团队本来就每天开会研究变化并调整计划，以适应这些变化。他们可以把这个支持任务当作他们遇到的任何其他问题或变化。在绝对不能等到下次冲刺而且产品所有者代表公司表示同意的极端情况下，他们可以把该支持任务加入冲刺积压工作表中，设置它的优先级（很可能是最高优先级），然后使用他们的计划工具（每日例会、任务板、燃尽图、用户故事、故事点，以及速度）让所有人了解情况并保证把任务完成。

一个产品所有者有这么大的决策权，与公司和客户联系如此紧密，每天还有大量的时间与开发团队一起工作，这怎么看都不现实。这难道不意味着 Scrum 那一套根本不可能做到吗？

不管你信不信，很多行业都有大量高效的 Scrum 团队，他们就是这么工作的。这是因为这些公司意识到，给予产品团队中的某个全职成员足够多的权威和发言权会带来巨大的好处，并且长远来看能够减少开发工作量。理解他们开发的软件的真正价值是每个开发团队的生命线，做到这一点是达到众多 Scrum 团队已经达到的理想效果的关键。那些有“惊人成绩”的团队几乎总是来自那些足够重视团队的工作并给他们分配合适的产品所有者的公司。

天下没有免费的午餐。把某个产品所有者分配给一个开发团队意味着让他放下手头工作，而且一般来说公司还是要给他支付相同数额的薪水。当一个公司相信 Scrum 时，他们的经理会看到这样做产生了效果，这种效果表现为该开发团队能够紧跟变化的业务需求。如果经理觉得这这笔买卖划算的话，他们会选择承受将产品所有者分配给该开发团队所带来的损失。

如果以上所说的这些在你工作的公司似乎都不太现实，那么这并未反应出 Scrum 的什么问题，这恰恰反应了你的公司对你的团队正在开发的软件的重视程度。很对企业明确告知他们的开发团队，不要每天拿开发中的软件相关的问题去打扰重要的主管和市场人员。这也是一种 Scrum 价值观与公司价值观冲突的最常见形式。

这也是为什么如此多的企业选择了效率更低的、需求先行的瀑布模型。为了减少市场人员和主管们与开发团队的沟通时间，他们宁愿让开发团队花更多的时间、精力和金钱去准备那些需求文档。天天跟开发团队打交道确实是很繁重的工作；对一个主管来说，看看文档，做点批改，然后拿出不那么优秀的软件，这样工作就轻松多了。公司常常可以通过增加业务分析人员来进行改进，让需求文档变得更好，还可以增加额外的质量工程师和测试人员来验证需求确实被正确地实现了。为了不打搅一个主管，付出的代价着实有点大，但这就是很多公司对软件开发的基本认识，而且，这种方法一样可以开发出优秀的软件（不过接下来你就会进入 CHAOS 报告中提到的那种状态：开发出不被使用的功能的风险大大提高了）。

如果一个企业认为主管和市场人员的时间和精力比开发团队的时间和软件本身更有价值，那么收入产出分析就很容易了，而需求先行也会最终成为开发软件的符合逻辑的选

择。如果你在这类企业里工作，那你接下来恐怕要先帮助你的经理和公司里的其他人理解 Scrum 和敏捷开发的价值。如果这么做没什么效果的话，那你至少可以先采用 Scrum 的一些实践方法，并开始以更好的（但不是惊人的）成绩来丰富你的履历。

我能理解冲刺计划中工作量估计完成之后的部分，但对于工作量估计到底是怎么回事还不是很清楚。到底应该如何估计任务的工作量呢？

估计任务工作量的方法有很多。Scrum 团队所使用的最流行的一种方法叫作“计划扑克”，这是又一个广受认可的 Scrum 实践。[9] 它由 James Grenning（敏捷宣言的起草人之一）发明，并通过 Mike Cohn 的《敏捷估计与计划》一书流行开来。具体方法如下所示。

> 一开始，每个参与工时估计的人都会拿到一副牌。每张牌上写着一个估计数字。比如，每个人都拿到写着 0、1、2、3、5、8、13、20、40 和 100 的一副牌。这些牌要预先准备好，而且上面的数字要大到可以隔着整个桌子都能看清。这些牌可以保存起来，在后续的计划会议上继续使用。
>
> 对于每个用户故事或主题，主持人阅读其描述。主持人一般是产品所有者或分析师，不过任何人都可以做主持人，因为这个角色并没有什么特权。然后产品所有者回答参与估计的人员的问题。
>
> 所有问题都回答完以后，每个人不公开地选择一张牌来代表他的估计。等所有人都选择完之后，大家再同时把自己的牌亮出来。
>
> 这时候大家的估计可能会非常不同。这其实是一个好现象。如果大家的估计不一样，估计较高和较低的人可以解释他们的估计。注意不要互相攻击，相反，要抱着学习别人是如何思考的态度进行讨论。
>
> ——Mike Cohn，《敏捷估计与计划》

计划扑克的效用在于帮助人们把知识、意见和想法用简单易懂的数字表达出来。牌面上的数字没什么讲究（比如，在过去，有些团队使用斐波那契数列中的数字：1、2、3、5、8、13、21、34、55、89），只是要挑选合适的间距以方便对任务做估计（比如：一个任务需要 13 个小时还是 21 小时？或者一个用户故事应该得到 5 个还是 8 个故事点？）。

跨国团队怎么办？

这是一个很好的问题，也确实是一个挑战。如果你的团队分布在世界各地的办公室里，这会带来很大的挑战。这些挑战并非无法克服，有很多敏捷团队虽然不在同一个地点办公，但一样开发出了优秀的软件。不过，当你减少面对面的沟通，转而依赖不那么可靠的沟通形式时，你总会面对更多因“传话游戏”而导致的问题。

解决这个问题的一个方法是把全球性的团队分成多个同城团队。除了各个子团队的每日

注 9：如果你想了解更多关于“计划扑克”的信息，我们推荐你阅读“计划扑克”的创建者 James Grenning（同时也是敏捷宣言的共同作者）的 *Planning Poker PDF*（https://www.wingman-sw.com/files/articles/PlanningPoker-v1.1.pdf）。

例会外，再增加一个由所有成员参加的大例会。会上所有子团队可以一起回答那三个问题。子团队根据各自的积压工作表进行自我组织，并把新的自我分配的任务带回团队。这样就给整个全球化团队一个属于它自己的、更大规模的计划以供每天进行检查。

这种大例会的方案并非大力丸，它并不能解决全球化团队所面临的全部（或者甚至大部分）问题。全球化团队面临的最大困难在于很难让每一位团队成员都对那些共同的高层次的目标保持积极性，正如我们在第 4 章中所看到的，这正是团队成员变成猪而不是鸡的主要原因。虽然不管你的团队是怎么分布的，肯定能够找到可行的方案，不过产品所有者要把整体的愿景传递给每个人是很有挑战性的，因为他不仅要面对很多个时区的问题，同时要使用那些跟面对面讨论相比效率更低的沟通渠道（像电话会议、电子邮件等）。

好吧，我知道每日例会可以让大家去处理那些正确的任务，可是即便是抱着良好初衷的开发人员有时也难免会去做一些不能充分利用时间的任务。Scrum 能避免此类问题吗？

不能。事实上，Scrum 的创始人 Jeff Sutherland 最近（在我们写作本书时）与 Scott Downey 共同发表了一篇题为“Scrum Metrics for Hyperproductive Teams”的论文[10]，说的正是这个问题。他们与很多 Scrum 团队一起工作并测量了一系列的指标，看看超高产团队到底有多高产。 他们发现，一个高效的 Scrum 团队交付项目的速度可以比不那么高效的团队最多快 400%。而要达到这种记录，保持开发人员不要偏离目标是关键的一点。

Sutherland 及其他 Scrum 思想领袖时刻在寻求改进 Scrum。上述论文诸多亮点中的一个是对每日例会的一个小调整。该调整的目的是让开发人员能够一致向着最有价值的目标前进，其对团队表现的影响也很可观。他们所做的调整就是让整个团队针对冲刺积压工作表中的每个条目回答以下几个问题（他们所要强调的内容），从第一个条目开始（我们姑且称这个条目为“一号优先条目”）。

- 针对一号优先条目，*我们*昨天做了哪些工作？
- *我们*在一号优先条目上已完成的工作量相当于多少个故事点？
- *我们*今天对于完成一号优先条目的计划是什么？
- 今天有没有什么正在或可能卡住*我们*或导致*我们*速度慢下来的事情？

整个团队要共同回答这些问题。与之前一个人一个人地说明自己的进展相比，现在大家要一起一个任务一个任务地把冲刺积压工作表过一遍，直到所有条目都过了一遍，或者达到了例会时限。

这样做为什么有效果呢？一个原因是它能让团队成员融入到每日例会中（而不是自顾自地说：“昨天我在写我的代码，我这两天都要继续写代码，没遇到什么障碍，完毕，耶！”）。同时，它让每个人都专注于那些最有价值的任务。一旦程序员开始专注于自身及自己感兴趣的任务，而不是专注于首先交付那些最有价值的功能，那么即使是高效的

注 10：Scott Downey 和 Jeff Sutherland 的论文“Scrum Metrics for Hyperproductive Teams: How They Fly like Fighter Aircraft”（该论文在 2013 年第 46 届夏威夷系统科学国际会议上发表，该会议于 2013 年 1 月 7 日至 1 月 10 日在夏威夷群岛的毛伊岛上举办）。

敏捷团队也会不经意地进入 CHAOS 报告所描述的那种境况。而上述几个问题则强迫整个团队保持警惕。

说到这，我们就得回到每日例会的本质，它实际上是一个正式的检验会议（就是你在大学的软件工程课上所学的质量检验），其目的是要改进团队的计划和沟通质量。只不过它恰巧还活泼有趣。

这正是 Scrum 的精髓之一：它是时刻处于进化中的；优秀的 Scrum 实践者总是可以贡献自己的力量，使 Scrum 变得更好。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 如果你还没有开始使用用户故事，那么拿一个你正在开发的功能，给它写一个用户故事。把它拿给团队中其他人看看，问问他们有什么想法。
- 现在就尝试一下计划扑克。与你的团队一起拿出一段固定的时间，并尝试估计你们当前项目中的几个任务。你可以自己打印你的扑克牌，也可以花钱购买［我们都是从 Mountain Goat Software 购买扑克牌，你也可以尝试他们的在线计划扑克游戏（http://www.planningpoker.com/）］。
- 估计完了之后，拿一个白板，或者在墙上贴一张大白纸，在上面画出燃尽图。你能坚持多久？注意看项目或迭代周期结束时燃尽图是什么样子。
- 你能不能通过分析你的上一个项目或者迭代周期并大致估算你们的项目速度？

更多学习资源

下面是与本章讨论的思想相关的学习资源。

- 关于 Scrum 和集体承诺：《Scrum 敏捷项目管理》，Ken Schwaber 著。
- 关于用户故事和广为接受的 Scrum 实践：《用户故事与敏捷方法》，Mike Cohn 著。
- 关于 Scrum 项目计划：《敏捷估计与规划》，Mike Cohn 著。
- 关于回顾会议：*Retrospectives: Making Good Teams Great*，Esther Derby 和 Diana Larsen 著。

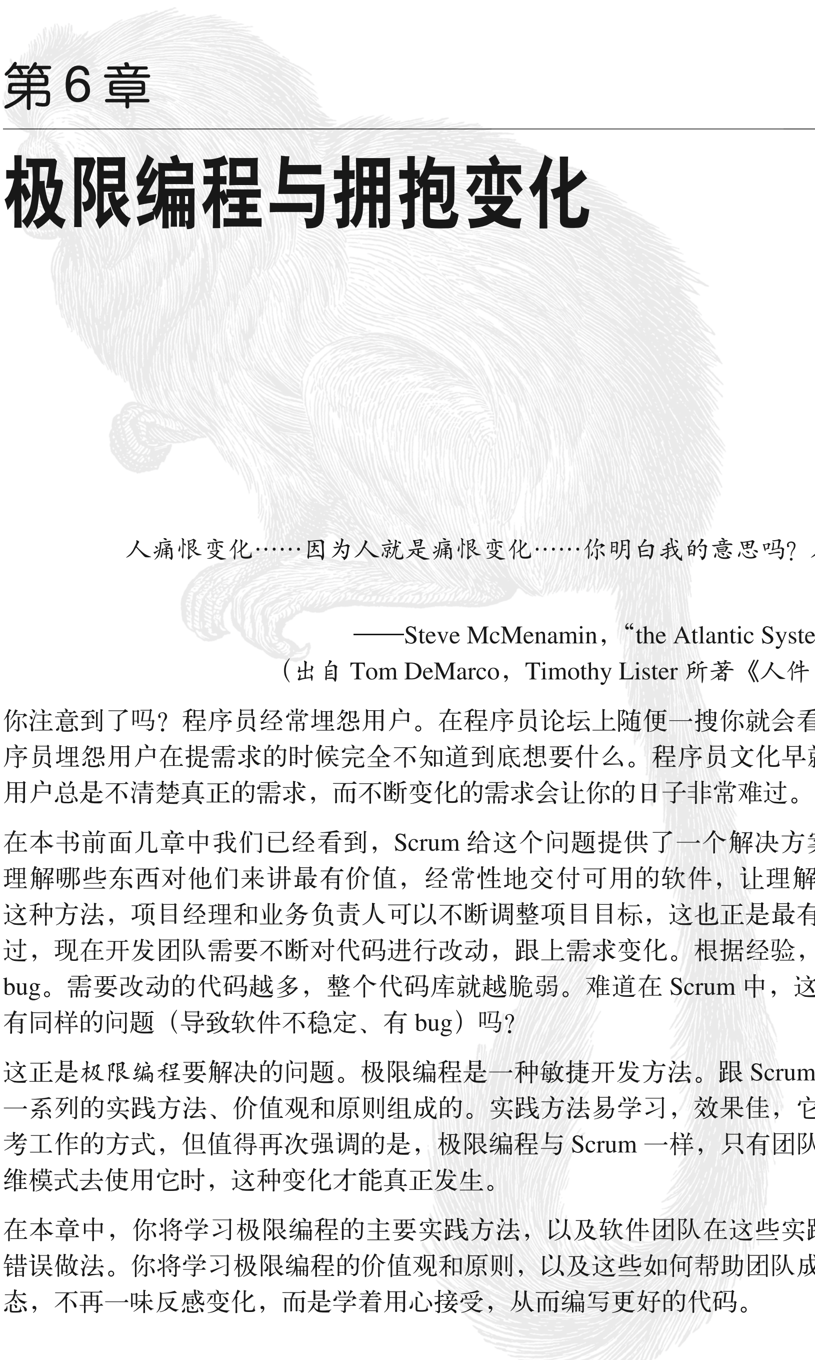
教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 教学过程中最大的挑战之一就是让团队真正理解集体承诺。多留意团队成员身上发生的这样一些情况：对于给出一个估计犹豫不决；没有真正参与到计划过程中；或者尝试让 Scrum 主管为他们做所有的计划。
- 跟单个团队成员聊项目的目标。留意那些视野狭窄、只关注所谓“属于”自己的某个功能或用户故事的人。鼓励这些人去做其他用户故事或功能的任务，并鼓励其他团队成员来做这个用户故事或功能。
- 指导 Scrum 主管，让他们把所有信息公开：把燃尽图和任务板挂在墙上；在一个显眼的区域召开每日例会。
- 要认识到办公室政治的存在。有时候，一些团队不愿意把项目真实的进展情况公开是由于他们之前有过这样的经验：经理发现项目进展不够完美时会大发雷霆。你的工作不是要帮他们解决这一问题，而是要帮助他们认识到：公司文化中有些地方需要慢慢改变，才能让 Scrum 的使用更加有效。

第6章

极限编程与拥抱变化



人痛恨变化……因为人就是痛恨变化……你明白我的意思吗？人真的痛恨变化，千真万确。

——Steve McMenamin，“the Atlantic System Guild”，1996，（出自 Tom DeMarco，Timothy Lister 所著《人件（原书第 3 版）》）

你注意到了吗？程序员经常埋怨用户。在程序员论坛上随便一搜你就会看到这类帖子：程序员埋怨用户在提需求的时候完全不知道到底想要什么。程序员文化早就接受了这一点：用户总是不清楚真正的需求，而不断变化的需求会让你的日子非常难过。

在本书前面几章中我们已经看到，Scrum 给这个问题提供了一个解决方案：与用户协作，理解哪些东西对他们来讲最有价值，经常性地交付可用的软件，让理解顺应变化。采用这种方法，项目经理和业务负责人可以不断调整项目目标，这也正是最有价值的东西。不过，现在开发团队需要不断对代码进行改动，跟上需求变化。根据经验，这常常会引入新 bug。需要改动的代码越多，整个代码库就越脆弱。难道在 Scrum 中，这样改动代码不会有同样的问题（导致软件不稳定、有 bug）吗？

这正是*极限编程*要解决的问题。极限编程是一种敏捷开发方法。跟 Scrum 一样，它也是由一系列的实践方法、价值观和原则组成的。实践方法易学习，效果佳，它们能够改变你思考工作的方式，但值得再次强调的是，极限编程与 Scrum 一样，只有团队成员以正确的思维模式去使用它时，这种变化才能真正发生。

在本章中，你将学习极限编程的主要实践方法，以及软件团队在这些实践上的正确做法、错误做法。你将学习极限编程的价值观和原则，以及这些如何帮助团队成员获得正确的心态，不再一味反感变化，而是学着用心接受，从而编写更好的代码。

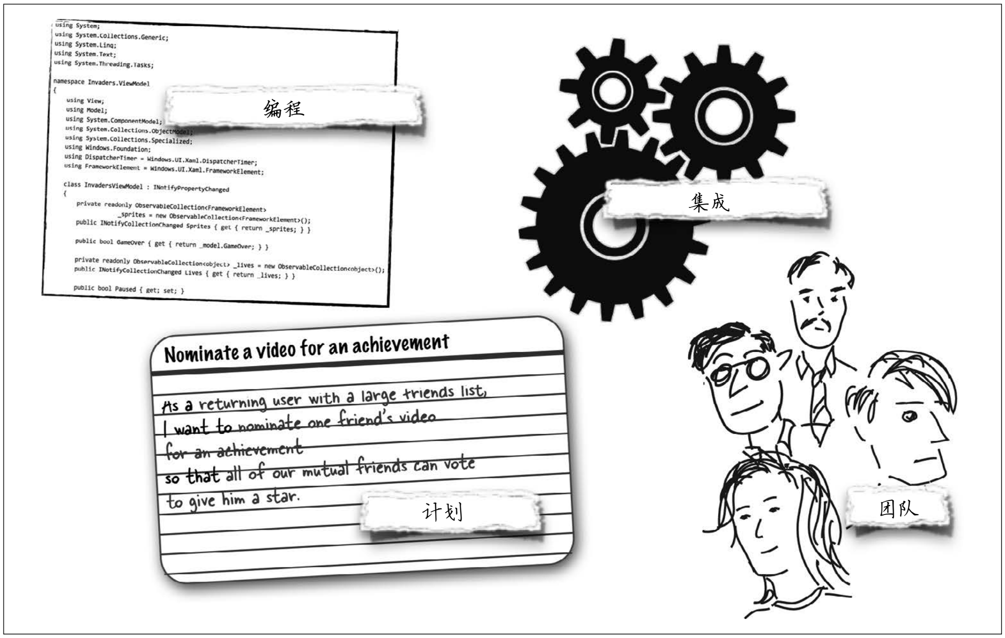


图 6-1：我们将极限编程的前 10 个主要实践分成了四个大类：编程、集成、计划和团队

故事：有一个正在开发虚拟篮球网站的团队

- Justin——一位开发人员
- Danielle——另一位开发人员
- Bridget——团队的项目经理

6.1　第1幕：开始加班

深夜在办公室加班时，Justin 心里总会有一种奇怪的感觉，虽然没喝多少咖啡，但还是有一种古怪的兴奋感。在家里加班的时候，他并没有这种感觉。

“又要开始一夜的奋战了，是吧？”Justin 的队友 Danielle 说道。Justin 和 Danielle 在大学的时候就是朋友。他们都是计算机系的学生，她比他高两届（他们学校的计算机系在全国数一数二），他们化学课上还一起做实验。他应聘开发在线虚拟篮球游戏软件的工作时，她帮忙做过内推。应聘的公司是 Chalk Player Online，这是一家开发体育网站的公司。她是团队里第一个欢迎他加盟的人。

可笑的是，Justin 并不想留到这么晚。倒不是说他不愿意加班。项目经理 Bridget 并不介意他在家工作，尤其是需要深夜加班的时候。可是不知怎么的，今晚他又在办公室呆待过了晚上 10 点，开始给女朋友发道歉短信，并开始请求室友帮他遛狗。

讽刺的是，Justin 和 Danielle 今天早些时候还说起要是晚上能准时回家该多好，但下午的时候 Bridget 就带来了坏消息：她和该项目的其他经理开了一个会，决定大幅修改代码。一开始，软件仅针对 NBA 的球队，但项目经理认为上线时把欧洲联赛也包含在内能够带来更多的收入。

而现在，Justin 和 Danielle 不得不再次深夜加班。

“这可太不公平了，”Justin 说，“三个月前，我们跟他们再三确认过了，说是只需要考虑 NBA 球队。”

Danielle 说，“不用你提醒我。那次会议本来就是我要开的。”

“他们最后还是要让我们把那些球队也加上。咱们不得不废掉大量的代码，还要对数据库进行大改。”他停了一下，接着说，“要我说，都怨 Bridget。我现在对她简直无语了。她根本不知道这么搞有多困难。”

Danielle 眼睛盯着 Justin 的背后，脸上满是惊恐的表情。

“Bridget 在我后面是吗？”

“是我，”Bridget 发话了，“注意，我很清楚这是一个重大变动。”

Justin 说：“并不仅仅是重大变动这么简单。问题是，我们已经告诉过他们开发方案了，而且是几个月前就告诉过了。他们也同意了。如果那个时候就告诉我们要考虑欧洲联赛，那我们的方案会完全不同。我们开发的系统中根本没有联赛这个概念，因为只有一个联赛，就是 NBA。”

“这是一个基本前提，”Danielle 补充说，“而改变这个基本前提会完全颠覆现有的系统设计。我们不得不废掉大量的代码，而这会带来很多问题。”

Bridget 问：“什么样的问题？”

“你用胶带和回形针修过汽车吗？”Justin 说，“就是像那样的问题。”

三个人面面相觑。未来还有很多个加班的夜晚在等待他们。

6.2 极限编程的主要实践

极限编程有 13 种主要实践，能够从软件开发的各个方面帮助团队编写出灵活多变的代码。与 Scrum 不同，极限编程的很多实践都是专门针对编程的，目的是要解决那些导致开发团队写出糟糕代码的问题的。事实上，正因为这些实践方法是专门针对编程的，人们经常错误地认为极限编程只有技术高手才能用。

在本章，我们会介绍极限编程的前十个主要实践。为了让你消除误解，更好地理解极限编程，我们将这十个实践归为四个大类：编程、集成、计划和团队。

6.2.1 编程实践

极限编程有两个主要实践意在帮助你写出更的好代码：测试先行编程和结对编程。这两项

实践的重点就是软件开发。先写测试代码，开发人员可以提高软件的质量，而结对编程（两个开发人员共用一台电脑）让每个人身边都多了一双眼睛，有助于减少 bug 进入生产代码的概率。

当一个开发人员采用*测试先行编程*（test-first programming）时［测试先行编程又称为*测试驱动开发*（Test-Driven Development，TDD）］，就意味着在他编写代码之前，一定会先编写一个自动化的测试。由于此时产品代码还没有写出来，这个测试肯定没法通过。而一旦这个测试通过了，那么该开发人员就知道他的代码可以工作了。这种做法构造了一种紧密的反馈机制，可以防止代码出现问题。先编写（无法通过的）测试，再编写代码让测试通过，定位问题并找出解决方案，然后再写下一个测试，如此循环。这些自动化测试通常被称为*单元测试*（unit test）。“单元”这个词很恰当：对于几乎每一种编程语言，代码都可以清楚地分解成各种单元（类、方法、函数、子过程、模块等），同时几乎每一种语言也都有至少一种构建和运行这些与相应单元对应的自动化测试方法。通过首先编写测试，开发人员可以保证每个单元都实现了其应该实现的功能。

测试先行编程可以确保每个独立的代码单元都正确工作，但是它的效用不止于此。它同时也可帮助开发团队避免一些最常见、最严重的代码维护问题。软件开发中时常出现这样的状况：代码的某个部分做了一个改动，结果一个似乎完全不相关的功能引入了 bug，因为改动的人不知道这两个功能有一个共同的依赖。当一个程序员编写了每次构建代码时都会自动运行的单元测试时，一旦共用依赖出现问题，这些自动化的测试马上就无法通过了。上述问题会立马暴露。开发人员发现问题的所在，这样的代码就无法进入代码仓库，问题不会变得更棘手。单元测试还可以帮助程序员编写易于复用的代码。例如，开发人员很容易写出结构不清的 Java 类：命名含混、初始化笨重以及其他容易出现的结构化问题。如果编写了使用该 Java 类的单元测试，这些问题就变得很明显了，而这时该 Java 类的具体代码实现还没有写出来，他可以很迅速地进行修改。之后，单元测试也变成了整个代码仓库的一部分，所以，其他开发人员可以通过它来了解该 Java 类的方法应当如何使用。

极限编程中另外一个针对编码的主要实践是*结对编程*（pair programming）。在使用结对编程的团队中，两个开发人员坐在一台电脑前进行编码。多数情况下，一个人敲代码，另一个人在旁边看，同时两人不断讨论。相比别的团队，对编程的团队引入的 bug 很少，因为总是有两双眼睛在盯着代码里。结对的另一个好处是比较放松，因为如果一个累了，那么另一个可以接手。结对编程的团队会讨论想法和方案，不断进行头脑风暴，因此也就更具创新。他们会彼此监督，遵守最佳实践。一个程序员完全可以做到偷工减料又不被发现，有个搭档在盯着，他这么做的可能性就降低了。我们曾从多个团队的开发人员那里了解到，结对的程序员相互协作（与他们各自单独工作相比）能够编写更多更好的代码。

6.2.2　集成实践

我们将两个极限编程的主要实践归入“集成”这个类别。第一个是 *10 分钟构建机制*（10-minute build）：开发团队需要一个自动构建全部代码的机制，而且完成自动构建的时间不超过 10 分钟。构建过程包括运行所有的单元测试并生成一个报告，说明哪些测试通过了，哪些没有通过。

10 分钟听起来好像是一个很随意的时间长度，但从团队的角度来讲这是有一定意义的。如果一次构建需要超过 10 分钟才能运行完毕，团队成员经常进行构建的概率就会降低。频繁构建对团队来讲是非常有价值的，因为这样可以让问题无处遁形。比如说，构建过程中运行了所有的单元测试，因此，构建完毕后，大家就清楚他们是否达到了自动化测试中所预设的质量高度。换句话说，10 分钟构建机制能够快速回答“我们的代码现在是否能正常工作？”这个问题。因为它的耗时足够短，大家都会经常进行构建，团队中的每个人都能够持续了解代码质量的最新状况。

另外一项集成实践是*持续集成*（continuous integration），这种实践主要基于一系列团队协作工具，允许多人在同一份代码上工作。也许团队中的每个人都需要随时修改同一份代码，但他们没法同时修改同一个物理文件，因为如果那样的话，大家就会不断覆盖别人的修改。因此，这里需要使用一个版本控制系统，该系统有一份中心的（或分布式的）主副本。每个开发人员可以*签出*（check out）该主副本，或者把整个代码仓库复制一份，仅供该开发人员自己使用［该私有副本常称为*沙盒*（sandbox）］。该开发人员可以在这份私有副本上工作，并周期性地将修改签入代码仓库。

这种方案的问题在于经常出现以下情况：当一个开发人员签入修改时，常常发现有人在她签出代码之后做了相冲突的改动。很多时候，这些冲突会在构建本地代码副本时显现出来，导致代码无法编译，有时候问题更加严重：这名开发人员签出代码至沙盒后有其他人向代码仓库签入了代码，冲突导致软件不能正常工作。将自己的修改签入代码仓库之前，代码仓库中的最新代码要*集成*到本地，而上述问题就会在这个集成的过程中显现出来。

这里就要提到持续集成了：开发团队需要不断地进行构建并注意编译错误或单元测试失败。很多团队会专门设立一个构建服务器，每隔一段时间自动签出仓库中最新的代码，运行自动构建过程，并在产生错误的时候通知开发团队。不过，设立一个专门的持续构建服务器只是持续集成的一个部分。持续集成意味着每个团队成员都要随时保持本地副本与主副本一致。也就是说，每个团队成员要定期将最新代码集成到自己的沙盒中。进行结对编程的团队有时候会使用一个实物*构建令牌*（build token），比如毛绒玩具或者橡皮鸭。这种东西看起来很幼稚，但这自有用处，你可以非常清楚地看到谁正拿着它。这个令牌在各组结对的开发人员之间传递，拿到令牌的那一组接下来需要将最近的代码集成到本地，修正出现的问题，然后把令牌传递给下一组开发人员。这么做能够保证集成过程中产生的问题及早发现并修正。

6.2.3 计划实践

极限编程团队使用迭代式开发方法来管理项目。与 Scrum 类似，极限编程中的计划实践也基于一个更长周期的长远计划，并会分解成一些较短的迭代周期。在*周循环*（weekly cycle）这项实践中，极限编程团队使用一周作为迭代周期，并紧密结合*故事*实践（等同于你已经学习过的用户故事）。每个周期以一个计划会议开始，会上团队成员回顾目前的项目进度，与客户一起挑选本次迭代的故事，然后将故事分解成具体任务，接下来对任务进行工作量估计并分配到具体的开发人员。

这些听起来应该很熟悉，因为这与 Scrum 的计划会议十分类似；事实上，很多极限编程团队都完全采用 Scrum 的计划方法（这也是第 2 章关于 Scrum 和极限编程结合的方法流行起

来的一个原因）。做完计划后，开发团队会先为选定的故事和任务编写自动化测试，然后用剩下的时间编写代码来让这些测试得以通过。与 Scrum 团队的自组织形式不同，有些极限编程团队会把当次迭代的所有任务放入一个列表，然后要求开发人员完成手头任务之后就接着去做列表中的下一项任务。这可以防止开发人员只挑自己喜欢的任务做，从而保证任务能够较为均匀地分配给所有人。

极限编程团队使用*季度循环*（quarterly cycle）这一实践来做长期规划。每个季度，整个团队会坐下来开会审视全局。会上将就一些*主题*进行讨论。主题就是宏观概念，可以实际帮助他们将项目的故事组织起来。对主题进行讨论可以帮助团队确定应该将哪些故事加入项目，并与该软件要解决的现实业务问题保持关联。大家也会讨论他们所经历的各种内部或外部问题，比如难缠的 bug，还有尚未落实的那些修正。他们也会花时间审视已经取得的进展：对用户需求的满足程度如何，项目整体的进展如何。有些极限编程团队也召开 Scrum 团队的回顾会议。

针对项目计划的最后一个极限编程主要实践是*丢车保帅*（slack），其实就是让团队将一些次要的、低优先级的故事添加到周循环。这些故事也会在计划会议上分解成具体的任务，但这些任务会留到整个迭代周期的最后。这样一来，如果开发过程中遇到了意料之外的问题，这些“车”就可以随时丢弃不做，留出时间让开发人员在迭代结束时依然交付完整的可工作软件。与所有的迭代开发方法一样，极限编程团队在迭代结束时只交付那些“真正完成的”软件，也就是说，该软件工作正常，所有测试都得以通过，并且可以演示给最终用户看。

6.2.4 团队实践

极限编程并不局限于编码。它也涉及团队协作。我们把两项实践归入了“团队实践”这一类别。第一个是*坐在一起*，顾名思义，不需要多做解释。大家坐在一起工作，便于沟通，工作往往效率更高。很多人没有意识到的是，个人编程时往往与外界隔绝，但作为团队的一员，开发仍然是一项高度社会化的活动。团队成员之间要不断互相请教，征求意见，并就可能出现的问题互相提醒。让大家一起坐在一个开放的工作空间里，自然可以鼓励这种沟通。关于工作空间到底要多开放有很多争论，因为开发人员要想高效率地工作也需要避开外界的干扰，况且很多程序员比较重视隐私，不希望自己的屏幕被过路的人看到。针对这一问题的一个流行的解决方案是“山洞空地相结合”的办公室布局。这一布局方案由 Steward Brand 在 *How Buildings Learn* 一书中提出[1]。在这种布局方案下,每个开发人员有自己私密的或共享的办公室（山洞），外面则连到一个较大的、有会议桌和结对编程用的工作站的公共空间（空地）。

另一个主要的极限编程团队实践叫作*大信息量的工作空间*（informative workspace），即对办公室做一些安排，使得在其中工作的人能够自动获得关于项目的重要信息。一个扩大工作空间信息量的流行做法是把大型的任务板和燃尽图安放在大家都能看见的墙面显眼位置，让大家一抬头就能看到项目进展，随时了解项目进度，由此作出更佳决策。随时可见、就在眼前的图表，以及其他用来在工作环境中展示信息的东西都称为*信息辐射体*（information radiator），因为它们会自动将项目当前的信息“辐射”给附近的每个人。

注 1：*How Buildings Learn: What Happens After They're Built*，Stewart Brand 著。

悬挂显眼的大图表并不是唯一的方法。当团队成员有疑问、遇到困难或指出问题的时候，他们会进行讨论。当这些讨论发生在一个共享的工作空间而不是在一个封闭的会议室中时，周围的人就会听到，进而了解相关事项的进展。团队成员自动接收到的项目相关信息，我们都称之为*渗透式沟通*（osmotic communication），不管该信息来自图表还是别人的对话。当然，这一类沟通既有它的价值，也有它的潜在问题。开发人员可能因为无意中听到别人的谈话而断了思绪，造成开发效率下降，因此，使用这类方法时需要把握一个平衡。高效的极限编程团队会找到平衡点，保证每个人都及时了解项目的现状。

为什么扩大工作空间的信息量这么重要呢？主要是因为这能帮助团队作出更好的决策，让团队把握如何推进项目，一如任务板对 Scrum 团队提供的帮助。关键在于，信息是属于整个团队的，而不仅仅属于团队领导或经理。信息应该是民主化的，应该不经任何过滤而传达给团队中的每个人。对项目信息的共享越多，团队中的个人也就更有能力对项目方向施加影响。

6.2.5 为什么开发团队抵制变化，上述实践如何提供帮助

没有人愿意仅仅因为提需求的人改变了想法而去修改自己已经写好的代码。在项目后期接到需求变更的消息可能导致你的不爽，或者让你感到受到冒犯，甚至可能导致近乎灾难性的后果：接下来的一个半月每个周末都需要加班，因为提需求的那个人让你和你的团队开发了一堆功能，但事前根本没有经过认真的考量。

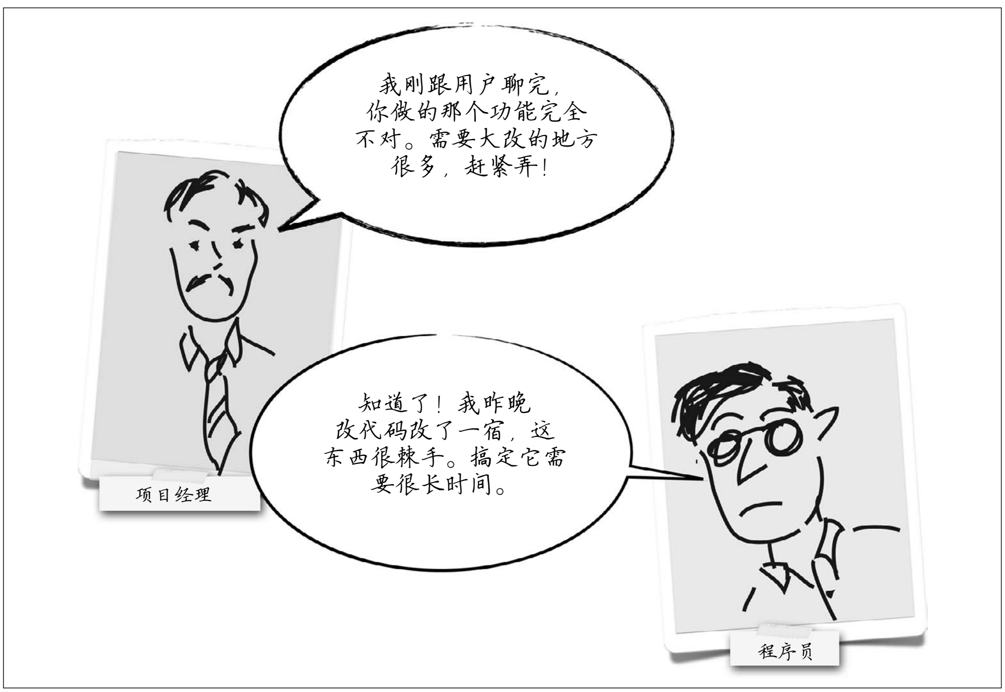


图 6-2：每个项目经理都经历过“需要做一个修改，但没法说服开发人员配合自己”的无助；而每一位开发人员也都了解“一个无关痛痒的小改动最后膨胀成了无底洞”的焦虑

这就是开发人员经常抱怨那些讨厌的用户总是善变的原因。项目做到一半的时候经常能听到这样的抱怨："等会儿，又要改？我要早知道的话，当初代码就不这么写了！你怎么就不能开始的时候就告诉我你到底想要什么？"

面对这样的抱怨，又有谁能够责备抱怨的人呢？开发团队之所以抵制这些改动，尤其在项目后期[2]，是因为这都是额外的工作量。而且不只是普通的工作，而是最坏的、最让人讨厌的那种工作。这些需求变动要求你回头拿起你原以为已经完成的东西，推翻它并重新来过。更要命的是，恼人的可不仅仅是修改代码。当初完成那些功能的时候，你的团队投入了大量的精力，仔细考虑了很多技术问题，并找到了你们力所能及的最优雅解决方案。现在有人让你把你的泰姬陵毁掉，因为他真正想要的是埃菲尔铁塔，真是要多气人有多气人。

不过，我们也要看到故事的另外一面。在项目后期提出这些修改的用户（或者经理）并不是有意要瓦解队伍的斗志。我们在本书前面的章节已经了解到，片面的视角会导致糟糕的结果，所以，让我们尝试从他的角度来看这个问题。他需要这个软件最终得以开发出来，这样他才能交差，而这也就意味着他需要与开发团队共同努力。现在他需要回答一些他不完全理解的问题，比如："谁有权限使用这项功能？""系统允许多少个用户同时使用？""从单击'开始'按钮到显示出结果所用的时间，多长可以接受？""用户需要先调整这几个选项，然后再作出那几个选择，还是反过来？"

会议本来是以用户解释他所面临的问题开始的，结果最后变成他得回答一系列看起来没完没了的技术问题，关键是他完全没法回答这些问题。时间紧迫，他感觉到如果他不能给出针对每个问题的答案，开发团队就会让项目延期，然后把延期的责任算到他的头上（实话实说，开发团队很可能真会这么干）。大家都知道开发软件是相当昂贵的，他又不是这方面的专家，所以他尽力给出他所能够给出的最佳答案，这样会议就可以结束，他也就不用再耽误开发团队的时间，也就免得耗费更多的开发资金。

这个问题到底怎么解决？哪一方是正确的？

敏捷方法（特别是极限编程）的优点在于，它首先承认我们并不完全知道我们具体要开发一个什么东西，而要想弄清楚这个问题，最有效的方法就是把它开发出来。使用这种方法的团队用可工作的软件说话，而不是用详尽的文档，因为从用户那里得到反馈的最佳方法就是先完成软件的部分功能，并把软件交付到用户的手上。

因此，与其让你的团队纠结于应付每一个变化，有没有可能创造一个环境，让团队可以做到优雅地应对这些变化，而不必经历无休止的加班与情绪化的冲突呢？没有人喜欢变化，但你也许可以找到一条路，使得每一个变化对项目的影响被限定在一定程度，这里不仅仅是针对代码的质量，也包括开发团队和用户的情感状态。

极限编程团队能够做到这一点。他们把这个叫作*拥抱变化*（embracing change）。一个极限编程团队并不会简单地把变化当成是不得不干的脏活累活；他们认识到要想给他们的用户做出最好的软件产品，唯一的办法就是经常性地获取用户的反馈并对这些反馈作出快速反应。他们知道会有变化，这些都已经写入他们的日程。变化不再是挥之不去的坏消息（也

注 2：请查看本书第 3 章中支持敏捷宣言的那些原则。那些原则中有没有适用于这里的呢？

不再是随之而来的追踪谁是遗漏需求的元凶）。进入极限编程的正确心态不仅意味着接受变化，更要求我们自己要求变化。

拥抱变化包含两个层面：项目范围层面和代码层面。变化对于代码的影响可能是非常严重的，有时候一个看起来应该很小的变化常常会令人意外地需要大量恼人且颇具破坏性的重写。开发人员常常会说“废掉这些代码”，然后把这个窟窿用“胶带、别针和口香糖”给它堵上。关于拥抱变化这项实践在代码层面的应用，我们会在本书第 7 章中探讨。这里我们将专注于极限编程团队是如何在软件的范围、功能和行为层面做到拥抱变化的。

那么，极限编程的主要实践是如何帮助开发团队做到这一点的呢？

- 计划
  当开发团队迭代式地计划并在每一个迭代周期结束时交付可工作的软件时，他们就可以不断获取用户反馈。团队中的每个人都知道他们是在主动要求反馈（即变化），而这也使得开发人员能够从情感上更容易接受变化，因为本来就是你自己主动要求的嘛。

- 团队
  身边有其他人帮助你的时候，问题会变得容易解决。同理，所有成员都坐在一起，团队解决问题也快得多。而信息量较大的工作空间也有助于大家思考相同的问题。这就给了每个团队成员以“控制变化以及变化何时发生的能力”，而不是对意外的变化猝不及防。这也让每个团队成员在作出会影响他们代码（还有生活）的决定时有了自主权。

- 集成
  意外变化所带来的最让人头疼的问题是，这些变化常常会导致整个代码仓库中的诸多部分都需要被推翻重来。假如咱们两个人是一个团队中的同事，你正在针对一个需求变化进行编码，这对我来说也有帮助，因为这可以让我早点知道有这么一回事，这也是为什么团队实践非常有用。不过，如果你对代码中某一个部分的修改导致了我这边的代码出现了一个意外的 bug，那我还是会猝不及防。而通过持续集成（由于我们的构建过程运行很快，所以持续集成也很容易），我们可以迅速定位问题并及早修复，从而将麻烦消灭在萌芽状态。

- 编码
  持续集成是如何帮助我们发现变化的？因为我们编写了一套单元测试，每次集成的时候都会运行这套测试，所以，当你集成你的修改时，也会运行这些测试，包括针对我写的代码的那些测试。如果其中某个测试没有通过，那么我们就可以协同工作（甚至结对编程）来修复我们代码中的问题。

如果上述这些实践都能够落实，那么需求的变化（即便是很大的变化）所造成的影响也就没那么大了。这有助于让团队意识到应对变化可以很容易。

要点回顾

- 极限编程团队所使用的**测试先行编程**，是指先编写描述产品代码行为的单元测试，然后编写产品代码以通过测试。这样可以形成一个反馈循环，用以防止缺陷的出现。
- 开发团队需要有一个 **10 分钟构建机制**，也就是一个能够在 10 分钟以内运行完毕的自动构建系统。
- 每个开发人员通过**持续集成**来不断地将其同事的修改集成进来，从而保证每个人的沙盒都处于最新状态。
- 极限编程团队以周循环或季度循环的方式进行迭代，并像 Scrum 团队那样使用故事。
- 开发团队通过将次要的、低优先级的故事加入迭代周期来制造一些可以在需要**丢车保帅**的时候丢弃掉的车。
- 极限编程团队坐在一起，他们通过渗透式沟通不时获取关于项目的信息。
- 极限编程团队在**高信息量的工作空间**里面工作，他们的工作空间里设置了**信息辐射体**（如，挂在墙上的图表）来自动把信息传达给附近的人。

故事：有一个正在开发虚拟篮球网站的团队

- Justin——开发人员
- Danielle——另一个开发人员
- Bridget——团队的项目经理

6.3 第2幕：计划有变，但我们还是看不到希望

六个星期前，Justin 本以为情况正在好转。他们埋头苦干，好歹算是完成了虚拟篮球网站的第一个 Alpha 版本，该版本完整地包括了 NBA 和欧洲联赛的球队。但是，问题依然存在。

Justin 记得 Danielle 曾经这样说，“把这称为‘问题’太轻描淡写了，这简直就是 bug 的地狱。”团队中的每个人都为这些 bug 感到挫折感十足，于是他们召开了一次大型会议，讨论代码质量问题。Bridget 在会上说：“咱们必须得拿出个方案来……马上！”大家纷纷提议加强设计和代码审查，拿出几个小时进行大型的测试，让所有产品经理同时使用软件，甚至还有人提议招聘一个全职的测试工程师。

这时，Danielle 向大家提到了极限编程。

“我们应该开始使用极限编程的方法，”她说：“这些实践将会解决我们代码的问题。”

团队中的每个人似乎都表示同意，同时 Bridget 也同意投入一些资源来帮助实施极限编程。她从业务部门那里为团队争取到了几个星期的时间来熟悉极限编程，而这将以一个为期一周的单元测试训练开始。由于他们原本的单元测试数目为零，因此整个团队利用一个星期

的时间编写尽可能多的测试。

可惜的是，Bridget 没能说服办公室经理允许他们坐在一起。不过，咖啡机旁边的一面墙是空的，所以 Danielle 在上面贴上了一个写有极限编程主要实践的清单，用来扩大信息量。清单所用的纸张和字体都很大。贴完清单后，Danielle 马上把在清单上的“大信息量的工作空间”一项前面画了一个对勾。当为期一周的单元测试训练结束后，她又在“测试先行开发”一项前面打了一个对勾。Justin 设置了一个构建服务器，该服务器每个小时从代码仓库中签出软件的最新代码，执行构建，运行单元测试，最后将结果通过邮件通知所有人。“持续集成”也可以勾掉了。

这些都是六个星期之前的事情。自那之后他们完成了大量的工作。可是，Justin 为什么还是感觉事情似乎没什么起色呢？

Justin 和 Danielle 刚刚跟 Bridget 开了个会。这个会开得不是很愉快。对玩家统计数据管理数据库的修正花费的时间比预期的要长，Bridget 对此十分不满。

整个团队召开全体会议之前，Justin 和 Danielle 私下碰了个头，两个人达成了共识：按时完工基本上是一个遥不可及的目标了。但是在全体会议上，当 Bridget 问到他们能否按时完工时，他们却给了肯定的答复，而且反复保证肯定能够做到。“这样，咱们过后再陪不是就是了。”Danielle 当时这么说。Justin 早就清楚，他们一直都在为了按时完成任务而寻找捷径，但这远远不够，他们肯定还是会延期的，但是他同意 Danielle 的看法，现在还不是告诉 Bridget 的时候。那样只会给项目带来更多的麻烦，从而导致更严重的延期。他不知道这种麻烦是不是导致极限编程无效的原因，不过清楚的一点是，极限编程似乎没有带来什么帮助。

现在 Justin 正在与 Danielle 一起享受他们的咖啡时间，他有点犹豫要不要提起这些问题，可是他又确实想知道 Danielle 是不是也有同样的感觉。“你觉得咱们的极限编程做得怎么样？”他问。

Danielle 的脸色不太好看。“我不知道咱们哪里做得不对。这些实践应该能解决我们的代码问题的，可是我刚才花了 4 个小时，才从我们开始使用极限编程之后所写的全新代码中定位了一个 bug。我们似乎在犯跟以前相同的错误。完全没有什么起色！”

“那么到底哪儿出问题了呢？”Justin 问。

Danielle 想了一下。“这个嘛，你看，现在已经没有人结对编程了，”她说，“大家一开始还都在结对，但过了没几个星期，就都觉得结对编程效率低。我倒不认为谁明确地决定说不再结对编程了。只是当结对的两个人中的一方休假时，这种结对关系就结束了。”

Danielle 说的确实是事实，而且是人所共知的事实。大家并没有明确地说要停止结对，但不知怎么的，大家确实不再结对编程了。测试先行？一样。Justin 还偶尔写几个单元测试，Danielle 则感觉她根本没时间去写那些东西。

事实上，就在几天前，Danielle 还在为放弃了结对编程而感到不快，所以她看了一眼 Justin 的代码。不看不知道，一看才发现，简直是一团糟。到处是注释掉的代码块；还有一团乱麻似的巨型函数。Danielle 并不怕直来直去，所以她给 Justin 指出了这些问题。“你从前写的那些优雅的代码哪去了？”

Justin 叹了口气，“我知道！我现在巴不得写出更好的代码，但是时间太紧了。我根本没有时间去思考。”

这一点 Danielle 是能够理解的，每一分钟都很宝贵，而 Bridget 则每天提醒他们绝对不能失败。所以，尽管他们两个人都很清楚自己可以做得更好，但时间太紧了，没有空闲打磨代码。

“你知道吗？这就是咱们不再结对编程的原因。修复那个比较大的生产环境 bug 之后，咱们就停止结对编程了，”Justin 说，“我当时不得不放下手头的工作转到那边，而你又不能干等着我，所以你就自己完成了当时咱们正在写的那部分数据库代码。我们当时就是没有时间结对。这些极限编程实践，没有一个是咱们有时间做的！”

Danielle 耸了耸肩。“可能是，也可能不是。”她沉默了一会，继续说到：“嗯……我能说句实话吗？我从打一开始就没有真正先写过单元测试。我就是像平常一样编程序，然后等写完之后再把测试加进来。不过我这么做应该也没什么影响吧？”

Justin 想了想，说：“这么说，我们的工作方式跟开始极限编程之前也没什么区别，不是吗？”

6.4 极限编程的价值观帮助团队改变心态

> 实践本身是空洞的。如果没有价值观作为它们的后盾，它们不过是机械的行为方式而已。拿结对编程来说，如果仅仅是为了结对而结对，那么它就完全失去意义了。为了讨好你的老板而结对完全就是一种折磨。反过来，如果是为了相互沟通、获取反馈、简化系统、发现错误、得到鼓励，那它就非常有意义。
>
> ——Kent Beck，《解析极限编程：拥抱变化（第 2 版）》

在关于 Scrum 的几章中，我们提到了 Ken Schwaber 的一句话，大意是说如果做不到集体承诺和自组织，那你就不算掌握了 Scrum。你也看到了一个团队是如何试着去实施 Scrum 但没能“掌握”这些（即照搬 Scrum 的做法，但没有实现自组织），最后只得到了有限的一点点提高。

极限编程也有一个很类似的模式。如果你和你的团队抵制变化，消极地抵抗用户修改软件功能的要求，而且认为构建容易修改的软件是不现实甚至不可能的，那么你就没有掌握极限编程的要领。如果你和团队没有理解简化（即简单的设计与复杂的设计的区别，简化是如何做到的，团队的文化和气氛是如何影响其写出简单的代码和架构的能力的，以及简化是如何防止 bug 产生的），那么你们就没有掌握极限编程的要领。本章后面很重要的一个主题就是拥抱变化。在第 7 章中，你将学习有关简化的内容以及它如何帮助你做出更易应对变化的设计。但在理解那些知识之前，我们应当回过头来，审视一下 bug 到底是从哪里来的。

那么，bug 到底是从哪里来的呢？

看看几乎任何一本软件工程的教科书，你就会发现上述问题的答案：推倒重来（rework）。[3] 诚然，bug 的产生有多种原因，但是 bug 被引入到软件当中（你可能在软件工程教科书中见过与此对应的一个术语：缺陷注入）的最常见的路径是通过对旧代码进行的重写。开发团队为实现某个功能开发了一个软件，但接下来有人希望对功能进行更改。于是开发团队不得不进行大量的重写：扔掉一些旧的代码，对已有的代码进行修改，添加新的代码，等等。这种重写过程很轻松地成为 bug 被引入的最常见来源。

说到这里有人会问，难道就没法避免这种修改吗？呃，理论上来说是可以的。而且软件工程教科书对此也提供了答案：将软件需求文档化。如果你真能做好文档化的工作（包括收集需求，将需求书面化，以及与软件的每一个可能的最终用户一起审阅这些需求），那么你应该能够避免上述修改，而开发团队交付的软件也更能满足“用户、客户和利益干系人的一切需求”。当然，即便这样也还是会多少有一些修改，但都是些小改动。

注意，上面说的都是“理论上”的情况。

其实，这种“理论上”可行的方法在现实中也是有可能做到的。很多团队都通过详实的文档开发出了大量优秀的软件。[4]

不过，这种开发软件的方法十分刻板，不够灵活，因为它要求编写软件规格文档的人在项目一开始就弄清楚全部（或者至少是足够的）用户需求。尽管这种方法在很多场合中是适用的，但它同样在大量的场合中是不适用的。因此，适用这种方法的项目需要变更管理系统，或者其他的机制来把需求的变化吸纳到现有需求中。

上述刻板的需求先行模式不太适用的场合，正是极限编程发挥其长处的地方。它能够帮助开发团队降低需求变动对现存代码的破坏性。重写和修改是可以存在的，甚至是受到欢迎的，因为代码在写出来的时候就考虑到了将来可能要进行的修改。

极限编程更值得称道的一点是，它避免了一个很大的问题，即：传统需求先行模式无法将真正优秀的点子加入到现有需求中。项目进行到中途的时候，有人冒出了很棒的点子，这种情况很常见。事实上，软件的早期版本会激发大家的灵感，引发头脑风暴。可是，在需求先行的框架下，这种头脑风暴往往会以糟糕的形式收场。请看下面的对话。

程序员：我有个好主意！

团队领导：确实是个好主意。但是这需要做大量的改动。你先把它写出来吧。等我们半年后梳理积压工作表的时候再重新评估一下。

程序员：可是，我只需要 10 分钟就可以完成这个修改！

项目经理：不行，风险太大。之前又不是没见过，即便是很小的改动，也有可能波及整个软件，导致很多 bug。别担心，我们不是否定你的想法。我们的意思是，现在还不是时候。

多气人啊。下一次该程序员冒出一个好点子的时候他会怎么做？很有可能缄口不语吧。

注 3：包括我们的第一本书，*Applied Software Project Management*。而且我们没有说错。传统软件工程和项目管理的思维是防止变化，而这与拥抱变化的思维是大不相同的。不过这种思维也是可行的，甚至并非与敏捷软件开发不相兼容。

注 4：我们就曾经做到过！对此我们丝毫不感觉到难为情。

畏惧变化的团队将会扼杀来自底层的创新。这倒不是说这种团队就没法开发出软件。他们能够开发出软件，甚至可能开发出优秀的软件。在需求先行的框架下工作，你能够达到“满足文档中的那些需求”这一目标，而这对很多团队来说已经很好了。但这里仍然存在问题。

6.4.1 极限编程帮助开发人员学会与用户协作

很多采用需求先行的瀑布开发模式的团队，其软件的第 1 版都很成功，而且第 2 版也经常是成功的。这并不奇怪。项目开始之前通常会有大量的讨论。具体的需求很明确，也足够紧迫，所以公司也愿意投入金钱和资源到项目。这意味着软件第 1 版往往反映了之前讨论过的所有内容，同时也汇集了所有人的好主意。其实，一般是相关的主题被不厌其烦地反复讨论，等到开发团队真正开始开发工作时，大家都感觉松了一口气。这是一种防止重写和变化的很有效的方法：因为大量的精力投入到了确认需求上，所以很可能这些需求不太会发生大的变化。另外，第 2 版包含的通常是那些在第 1 版中没有时间实现的有价值的功能。因此，需求先行的团队在项目初期的成功并不鲜见，他们的第 1 版软件也常常工作正常且不需要太多的修改。

问题出现在项目的第 2 版、第 3 版以及第 4 版中。这时候已经有用户在使用该软件了。我们已经知道，交付可用的软件是引发用户反馈的很好方法（这也是为什么敏捷团队更喜欢可用的软件而不是详尽的文档）。现在用户提出了一些很好的反馈意见，而多数时候要实现这些反馈意见，就必须要进行重写。

那么，如果开发团队编写的软件很难进行修改，会发生什么呢？他们会走一遍计划流程，评估这些用户需求，最后判定实现这些修改有很大的风险会给现有代码引入 bug，进而导致这些修改需要大量的时间才有可能实现。这样看来，他们对变化的畏惧也就非常合理，也理所应当了。同时，这也强迫用户、客户和利益干系人需要找到一种修改需求的方法，使得这些修改能够在现有代码的基础上实现。用户与开发团队间在软件第 1 版中所建立起来的良好伙伴关系很轻易地就变成了一种对立关系，而这种关系把对需求的挖掘变成了对合同条款的讨价还价，同时也让获得客户的配合变得更加困难。

极限编程给开发团队提供了一条避开上述问题的出路，即：帮助开发团队把代码变得易于修改，前提是开发团队能够以正确的思维方式实践极限编程（在第 7 章，你将了解更多这方面的内容）。

极限编程还会改变团队中每个人对于质量的认识。质量并不仅仅是避免 bug。它更关乎满足用户需求，即便这个需求跟用户最开始要求的不一致。传统的需求先行团队把功能设计（软件能做什么）和技术设计（如何实现该软件）分开来对待。软件的功能被写进一个规格文档，然后交由开发团队予以实现。如果开发人员有疑问，或者功能看起来不正确，他们可以去找编写规格文档的人，让他予以解释和澄清。这就开始了一个“传话”游戏：编写规格文档的人去询问用户、经理，以及其他当初提供了需求的人；然后他再把这些人的话“翻译”成程序员能够理解的语言。有意思的是，需求先行的团队认为这种沟通方式效率很高，因为它不需要“浪费”开发人员的时间去跟用户沟通。（大家都知道程序员不善交际，不是吗？）

极限编程的秘诀在于，它强迫开发人员像用户那样去思考，而这要从与用户沟通开始。（什么?！）如果开发人员不具备与用户沟通的能力，那他们就得开始培养这种能力。极限编程的实践其实会有助于沟通能力的培养。拿测试驱动开发来说，开发人员需要首先编写测试，这能够防止 bug 的出现。同时这也可以强迫每一个开发人员在写下一段代码之前去思考编写代码的目的。 如果习惯写每行代码前思考功能需求，那么程序员在每个周循环的计划会议上也会提出相同的问题。

极限编程还可以防止需求先行框架下消极的自我保护。在需求先行的框架下，一个开发人员可以完全按照规格文档逐字逐句地进行实现，完全不用去考虑规格文档中所写的内容是否真正解决了用户的需求 。但是如果这个开发人员养成了写代码之前思考功能的习惯，他就会开始发现问题，而且会自然而然地要求得到回答，因为他写测试的时候需要知道问题的答案。他不会把责任推卸给撰写规格文档的人。当一个极限编程团队的每一位成员都做到这一点的时候，他们会不断地从用户那里寻求解释和澄清，尽最大努力去理解用户需要的到底是什么，以及软件的功能应当如何实现才能帮助用户完成他们的工作。他们直接与用户接触，而不需要通过一个中间人，这能够节省很多时间。

换句话说，需求先行能够帮助你构建“预想中的”软件，而极限编程则帮助你构建你的用户真正“需要的”软件。使用极限编程能够让每个人都开心：团队成员更开心了，因为他们能够把时间花在解决真正的问题上，而不是花在跟代码较劲上；用户也更开心了，因为他们得到了自己想要的软件（而不是他们半年前以为自己想要的软件）。这个过程形成了一个更高层面的团队，包括用户、客户以及各利益干系人，他们不必刻意要求自己预知半年甚或一年后的需要，而可以在能力范围内专注于开发目前看来的最佳软件，解决眼下所知的最重要问题。

那么，极限编程是如何做到这些的呢？

6.4.2　开发团队的怀疑会破坏实践的效用

极限编程的实践可以帮助你在写下大量的代码之前专注于代码的质量。Scrum 的作用在于让你的客户了解你能够做到什么和你做不到什么；而极限编程的作用则在于让你能在修改代码的时候尽可能少地引入 bug。它还能够帮助所有人建立对质量的追求，这种追求会渗透项目的所有开发工作。

这对很多技术人员来讲是一种新颖的思维方式，哪怕他们经验丰富，出类拔萃。如果技术人员感觉不靠谱的计划给项目带来了问题，那么要让他们认可 Scrum 并不困难，尤其是那种“聊胜于无”的 Scrum。Scrum 也确实是改进开发团队进行项目计划的一个有效手段。可是，当一个开发人员面对极限编程的实践，并发现这些实践需要改变编码方式的时候，会发生什么呢？

关于软件开发的一个最基本的事实是：编写一个程序的方法有无穷多种。让两个开发人员去编写同一个代码单元（类、函数、库或者适用于某种语言或框架的任何其他单元），他们写出相同代码的概率非常低。那么其中的一个人写的代码比另外一个人的质量更高，还是什么难以想象的事情吗？



图 6-3：对极限编程的一个普遍反馈是：理论听起来挺好，但是像结对编程和测试先行开发这类做法“在我们的团队里就是行不通”。这倒不一定是对这些做法的一种反对，只是觉得这种改变很困难

显而易见，这跟编程技能有很大关系。可是技能并不是全部。事实上，很多拥有非常优秀的开发人员的团队，同样在维护着糟糕的代码。这些糟糕的代码常常是由于开发人员赶着做一些计划之外的修改而导致的。最终的结果就是大量的 bug，还有弥漫在团队中的对修改和重写的畏惧。

这是一个导致开发团队投向极限编程阵营的常见原因。这其中的很多团队（尤其是他们的经理）更多地把注意力放在极限编程的那些专门致力于发现和预防 bug 的实践上。结对编程是个不错的主意，因为它相当于有了不间断的代码审查（多一双盯着代码的眼睛就意味着更多的 bug 会在进入代码仓库之前就被揪出来）。测试驱动开发意味着开发团队会编写并维护测试，用来发现开发过程中引入的 bug，而持续集成则意味着这些测试会一直运行。这一切听起来都很美好，不是吗？这么做肯定能够发现更多的 bug。

一个通过极限编程得到“聊胜于无”效果的团队，也会发现结对编程能够防止引入 bug，也知道测试先行开发和持续集成能够及时发现引入的 bug，但是这种团队常常会认为这些东西有固然好，但没有也行。对他们来说，完成项目的最低要求就是写完所有的产品代码。而当项目进度滞后的时候，他们就会倾向于只去完成这个最低要求，而把结对编程、测试先行这些“华而不实”的东西抛到一边了。他们会这么说，“我们的进度已经落后了，所以我们没有时间去编写单元测试”，或者“要是我们能给每个任务分配两个程序员当然很好，但是要想按时完成项目，我们确实没有那么多的人手。”

这种思维在初次接触极限编程的团队中并不少见。这种思维的问题在于，它把那些非常好的实践当成了像节食或锻炼这类东西：好固然是好，要是我的生活不这么忙，我肯定去做了。即便对那些严格执行这些实践的人来说，如果这些实践感觉上不是必须的，那么当日程很紧或者项目遇到问题的时候，他们也会把它们抛到一边的。

这也是为什么把极限编程实践当成是一个等着打勾的清单（更有甚者，根据采用了多少极限编程的实践，得出一个“极限编程的比例”），不但不会提升效率，反而会对团队的生产力有负面影响。这就像是把极限编程的实践当成了最终的目的，而不是为了达到某个目的的一种手段。这就像是一个巨大的警示牌，说明这种团队还没有真正掌握极限编程的精髓。

像这种尚未掌握真正的极限编程思维的团队，就必须要改变他们的思维方式。这种思维方式的转变对一个团队来讲可能是件了不得的大事，尤其是当他们觉得这些实践不值得花费他们的时间时，更是如此。那么，如何才能让开发团队把极限编程的实践当作是开发优秀软件的“头等大事”来抓，而不是把它们当作是锦上添花但却可有可无的东西呢？

6.5 正确的思维从极限编程的价值观开始

拥有正确思维的团队总是会去采用优秀的实践，完全不需要别人的督促，因为他们清楚那些时间会让他们开发出更优秀的软件。与 Scrum 类似，极限编程也有五则*价值观*。这些价值观也同样能够帮助你的团队真正接受极限编程，并避免“聊胜于无”的结果。极限编程的价值观帮助团队意识到，那些优秀的实践是通往优秀软件的显而易见的道路。反过来，所谓掌握极限编程的正确思维其实就是说，要理解极限编程的那些实践是如何让你能够写出更好的代码的。

6.5.1 极限编程的价值观

- *沟通*
  每个团队成员都清楚其他人在做什么。
- *简化*
  开发人员尽量让写出的代码简单、直接。
- *反馈*
  不断进行测试和反馈，以保证产品的质量。
- *勇气*
  每个团队成员都应该专注于为项目作出更佳的选择，即便这意味着不得不抛弃失败的方案转而从不同的角度去解决问题。
- *尊重*
  每个团队成员对项目都是重要的、有价值的。

这些都是听起来很美好的抽象概念。怎么可能有人不同意呢？

（你很快就会发现，也许你自己就不同意……不过不要紧，只要你能保持一个开放的态度就行。）

6.5.2 以善意铺就

一个团队刚开始尝试极限编程，每个人都很乐观。他们清楚他们的代码质量存在问题，他

们为 bug 感到头疼，而且看来似乎极限编程能够最终解决这些问题。

接下来，走出理想，走进现实。直觉上，在思考问题并编写代码的整个过程中，让一个人全程坐在你身边似乎非常昂贵。如果说结对编程的唯一好处就是多一双眼睛来发现 bug 的话，那么有其他更经济的办法来达到这个目的。代码写完之后进行一次代码审查很可能会发现一样多的 bug（结对编程后也没什么理由不做代码审查）。但是如果你面临着截止时间的压力，你可以很容易地说服自己，认为代码审查比结对编程也差不到哪去。“我这也算是结对吧，因为有另外一个人也参与了！”

几乎每一个极限编程的实践都有一条类似的捷径。你们不必坐在一起，每天开个会就行了；你也不用要求团队成员不断地在他们各自的沙盒上进行集成，弄个服务器自动做集成就行了；你不必先写测试，等写完代码之后再写测试也一样，依然能够在测试通不过的时候发现 bug。

上面说的这些都对。而且在每一种情况下，都比什么都不做要强，于是你得到的就是“聊胜于无”的结果。

让我们回顾一下本章开头 Kent Beck 的那句话：“实践本身是空洞的。如果没有价值观作为后盾，它们不过是机械的行为方式而已。”这与 Ken Schwaber 针对 Scrum、自组织和集体承诺所说的话很相似：每个极限编程实践都不仅仅是简单地多一双盯着代码看的眼睛，或让大家坐在同一个房间，或给代码增加测试；更多的是帮助团队把极限编程的价值观带入到项目中去。

这就是为什么那些看似与极限编程的实践有着相同作用的捷径只能带来“聊胜于无”的效果。它们改变的是人们做事的技术性部分，但是它们并没有改变大家的意图或者思维方式。

刚开始使用极限编程的团队经常会跳过价值观，因为实践似乎更有吸引力。很多开发人员真的会打开一本极限编程的书，快速地看一眼极限编程的价值观，然后直接翻到有关实践的章节去。

实践是具体的。你可以想象自己如何实践，同时你也了解它给你的项目带来的影响。另外，实践还是孤立的，你可以把某一项实践加入到你的项目中，而不必改变你对你和你的团队所做的所有事情的想法。

价值观则没有那么容易理解。它们更加抽象，而且它们会影响到这个项目以及每个人看待项目的方式。技术层面上，你可以把实践加入到项目中，而不必理解价值观（这将带给你“聊胜于无”的效果），但是你没法在完全没有实践经验的情况下理解价值观。

这并不是极限编程独有的现象。Scrum 也是如此。一堂典型的 Scrum 入门培训课常常只有一两张幻灯片是讲 Scrum 价值观的，讲解的时间不过几分钟，然后马上就进入所谓的“干货”部分，即具体的实践。但在这一点上我们不应该对培训人员要求太苛刻。与通过价值观来改变人们的思维方式相比，教会他们使用极限编程或 Scrum 的具体实践要简单得多。大多数培训人员的经验是，如果他们讲一堂专门针对价值观的课，来自开发人员的反馈往往是：课程“太理论化了”，没有什么他们能够在项目使用的实用信息。同样，这也是可以理解的反应，在你真正理解那些价值观之前，很难看到他们的实际效用，而且没有价值观支撑的实践无法真正起作用这一点也不是一目了然的，这就成了一个鸡生蛋、蛋生鸡的问题。

回到 Justin、Danielle 和他们的虚拟篮球项目上。他们没有花时间去理解极限编程的价值观，就迅速将极限编程的实践用了起来。虽然你并不了解他们每天的工作细节，不过你可以尝试分析一下，看你是否能够发现他们的极限编程尝试哪里做得不对。下面是我们发现的几个例子。

- 沟通
  Danielle 和 Justin 并没有讨论过要不要继续使用他们原来以为他们所需要的实践，而且他们肯定没有跟 Bridget 谈过工作的真正进展情况，这对 Bridget 来说无所谓，因为如果一旦项目出现问题，她完全可以责备他们没有提前告知她。

- 简化
  Justin 的代码越来越复杂，甚至令人费解。他其实有能力写出更好的代码，但是他就是感觉自己没有足够的时间去那样做。

- 反馈
  从 Justin 和 Danielle 停止结对编程开始，他们两个人就基本不怎么碰头了。如果 Danielle 早点看一看 Justin 的代码并给他一些反馈意见，也许现在不会这么糟。另外，虽然工作空间有一定的信息量，但这些传递出来的信息并不能帮助大家就有关项目的问题作出更好的决策。知道团队使用极限编程的程度并不能提高软件的质量。

- 勇气
  当团队“几乎不可能”按时完工时，向大家说出实情需要很大的勇气，尤其是说话的人可能会招致埋怨的时候。Danielle 和 Justin 不具备这种勇气。

- 尊重
  在专业场合，没有什么比要求团队按照一个不可能的时间表去交付项目更加不尊重人的了[5]。Bridget 不断地设置过于激进的截止时间。更糟糕的是，她并不认为自己需要与团队同舟共济，比如那次她在周五召集了一次会议，要求团队在周一早晨之前完成一大堆修复工作，然后让开发团队周末加班，自己却休假去了。（本章前面并未提到这一点，但是我们得到了确切的消息，此事的确发生过。）

当极限编程的价值观没有被每一位团队成员内化时，他们有可能得到“聊胜于无”的效果（最好的情况也就是这样）。他们会说有一些实践（最常见的是结对编程和测试先行开发）“在我们的团队中就是不起作用。”团队成员倒不会真的去反对那些实践，他们只会表示这种改变会很困难，于是更倾向于那种清单式的方式，而且清单要以其他实践开始。

一个接受了极限编程价值观的优秀极限编程团队具备正确的思维。而一旦他们做到了这一点，团队成员对极限编程的看法也会发生有趣的变化：他们开始理解那些具体实践的意义了。

比如，一个团队刚开始实施持续集成的时候常常是这样的：很多团队一开始会躲着它，因为他们认为持续集成需要他们设置一台构建服务器（这需要花时间，而且需要专门分配一台电脑给它）。但实际上持续集成可以通过使用一个简单的“构建令牌”（像一个傻傻的玩具或者毛绒玩偶）就能够实现。想象一下一个刚接触极限编程的团队采用这种方法。第一

注 5：讽刺的是，有些经理自欺欺人地以为有压力才有动力。（原书编者注：他们最终也是自食其果。）

个拿到令牌的人花几分钟时间把版本管理系统中的代码集成到她自己的沙盒中，进行必要的测试以确认程序依然能够工作，然后她把代码签入版本控制系统中。令牌接着逐个开发人员地传递过去，轮到的人就将代码集成到自己的沙盒中，进行测试，签入代码，然后把令牌传递给下一个人。

这么做对一个团队学习极限编程的价值观有什么帮助呢？

构建令牌经常会反复地停留在同一个人的手中很长时间，因为那个人认为他不能中断手头的工作。团队中的其他人则需要想办法跟他沟通这个事情，让那个人理解持续集成对团队的重要性，这将有助于提升他们的沟通能力，并让他们对沟通能力更加重视。而那个固执的开发人员则能够学会“尊重”这条价值观，因为无论他手头正在做什么重要的工作，他都需要把团队的持续集成这件事优先做完。通过持续集成，可以学习到有关极限编程价值观的其他方面。比如反馈（今天进行集成发现了一个大问题，于是为后面节省了时间）；勇气（如果发现的这个问题是他人的错误导致的，你得给他们指出来，这可并不总是件容易的事，尤其对内向的人）；如此种种。

这就是采用某一项极限编程实践对塑造团队的思维方式所能够带来的帮助。

不过，如果你的团队已经具备了与极限编程相兼容的思维，你就不必从采用某项实践开始学起了。你也可以问自己以下问题。

- 对于已经写出来的程序，由于发现其不能工作而把它推翻重来，你的团队对这种做法是否认可？你的老板呢？
- 如果老板要求的工期很紧，开发团队是否真正相信编写单元测试是按时完工的最快途径？即使这意味着要编写更多的代码。
- 如果一个资历较浅的程序员接手了某项任务，团队中其他人是否愿意给予他足够的话语权让他完成该任务？如果他们不同意这个程序员的解决方案呢？如果有人觉得自己可以做得更快呢？是否允许该程序员失败并从他的失败中汲取教训呢？

当你的团队还没有完全接受极限编程的价值观时，你能做些什么呢？

故事：有一个正在开发虚拟篮球网站的团队

- Justin——开发人员
- Danielle——另一个开发人员
- Bridget——团队的项目经理

6.6 第3幕：势头的变换

“Justin，刚发生了一件很奇怪的事情。”

Justin 正在沉思关于他那部分代码中以多项统计数据给玩家排名的算法。他停下了手头的工作，抬头看着 Danielle。

“还记得咱们当初结对编程效果不怎么样吗？”

“记得啊，”Justin 说：“先是我写代码你在那盯着看，然后咱们互换。盯着看了半天，但是没什么帮助。事实上，我当时算了算咱们俩每个小时写出来的代码的行数，基本上工作效率降低了一半。我觉得可能正是因为这个咱们逐渐就不再结对编程了。”

“嗯，我刚才跟 Tyler 一起结对编程来着，感觉非常有意思。”Danielle 说。

“Tyler？那个新来的小子？他好像六个星期前才刚从大学毕业。跟他结对能有什么意思？”

Danielle 点了点头。“我也挺意外的。我跟他结对编程不过是因为我觉得这样可以让他尽快熟悉一部分代码。但是当我们一起过了一遍玩家数据缓存相关的代码时……”

Justin 打断了 Danielle 的话。“啊！写那部分代码的时候可真是折磨人啊。”

“我知道。你猜怎么着？ Tyler 问我为什么我们当时没有把关键字和散列值与玩家对象存在一起。”

Justin 瞪着眼睛坐在那，过了几秒钟，他的下巴才开始放松下来。系统中有一个数据缓存模块，当初是他和 Danielle 为了解决某个严重的性能问题而开发的。那个模块相当复杂，所以他花了一分钟时间才想明白当初他们是怎么存储的数据。“等等！喔，我晕。也就是说缓存中会有垃圾数据。”

“正是。”Danielle 说。

“是 Tyler 发现的？那个新来的小子？”

“没错。我们做了一个小型的测试，发现这是一个大问题。Tyler 指出了问题之后，倒也不难修复，不过如果我们现在没有发现的话，将来肯定是个大麻烦。我们所能看到的就是，过不了多久，一个玩家的统计数据就会出错。我们差点就把这个 bug 发布到生产环境中去了。”

Justin 曾经打印出了一份极限编程的价值观，而且把它贴在了格子间的墙上，之后扭头就把它给忘了。他指了指那张纸，说：“说老实话，我一直没怎么注意这些东西。不过我觉得我刚刚学到了一些有关沟通方面的东西。”

“还有尊重，”Danielle 说，“将来我肯定还会征求 Tyler 的意见，而且我想继续跟他结对编程。”

6.7 理解极限编程价值观，拥抱变化

在极限编程的价值观与它的具体实践之间，存在着一定的距离。价值观很宽泛，它们给你提供了一种对协同工作的思维方式，但是对极限编程的初学者来说，有时候很难把它们应用到具体的实践中去。

幸运的是，极限编程有一套*指导原则*，可以指引你在实际项目中应用这些实践。这些并非唯一的原则，但它们为很多极限编程团队所采用。

下面是这些极限编程指导原则的列表。这个列表看起来很长，但是不要忘了，极限编程并不要求你记住这些原则。极限编程的价值观很宽泛，所以这些指导原则更多描述的是关于极限编程团队中的人们如何看待问题的细节。所以，这些指导原则对于你确定自己的团队

是否具备正确的极限编程思维有很重要的作用。

- 人性化
  牢记软件是人创造出来的，要平衡团队成员的需要与项目的需要。
- 经济因素
  软件的背后总是有人要掏腰包的，每个人都要考虑到预算问题。
- 共同利益
  寻找那些能够使得个人、团队和客户都能受益的实践。
- 自相似
  一个月度循环与一个周循环是一样的，与一个日循环也是一样的。
- 改进
  今天尽你的全力，同时要知道你明天怎么能够做得更好。
- 多样性
  大量持有不同意见和视角的人在一起工作，从而得出更好的解决方案。
- 反思
  优秀的团队会持续反思他们软件开发过程中哪些做法有效，哪些做法没有。
- 流畅
  不断地交付意味着连续的开发工作流程，而不是明晰的阶段性流程。
- 机会
  团队碰到的每一个问题都是学习关于软件开发的新东西的一个机会。
- 人员冗余[6]
  即使乍看起来有点浪费，人员冗余确实能够避免大的质量问题。
- 失败
  你可以从失败中学到很多东西。应该允许失败的尝试。
- 质量
  降低质量标准并不能让你更快速地交付产品。
- 责任明确
  如果某个人对某件事负责，那么他应当有足够的权威来完成它。
- 慢慢来
  向着正确的方向缓步前进，在采用新的实践时，不要太过大刀阔斧。

6.7.1 极限编程的指导原则

这些指导原则有助于你理解某些极限编程中用到的具体实践，当你开始去探究实践时，它们又帮助你理解这些指导原则。

注 6：指结对编程中两个程序员一起工作。——译者注

与极限编程价值观的情况类似，刚接触极限编程的人总是禁不住草草地看一眼极限编程的指导原则然后直接跳到实践部分。人们之所以更愿意走这样一条路是有多重原因的。你可以把某项实践加入到项目中，而不必承认项目存在问题（“我们做得挺好的，不过我们可以做得更好！”）；如果你跳过价值观和指导原则而直接进入实践的部分，你可以很容易地采取那种“清单式”的办法，把所有实践都写在一张清单上，然后把那些已经为团队所执行了的实践勾掉。如果你的目标是所谓“完全地”采用每一项极限编程实践，那么这种做法完全可行。

但是，如果你的最终目标是帮助团队开发出更好的软件，那么这种清单式的极限编程一定会失败。你能得到的最好结果也不过是“聊胜于无”，而随着时间的推移，这些实践极有可能会逐渐被遗忘，开发团队还是会回到他们之前的老路上。这在极限编程和 Scrum 团队中都是常见的现象，在现实世界中我们也接触过很多亲身经历过类似状况的人。团队成员一般会把这归咎于方法本身：“我们下了很大的力气编写测试（或者结对编程，或者每日例会，等等），可这并没有给我们带来什么帮助。这种方法肯定不好用。”

指导原则能够帮助我们理解为什么会这样。指导原则强调的是自我反思。它们强迫你跳脱出来，真正的开始思考你和你的团队是如何工作的。当你真正花时间去理解价值观和指导原则时，你就会开始发现你的团队哪里存在问题。这当然让人不好受，也常常是一种负面的体验。比如，你可能通过“失败”这条原则发现你的团队文化中是不容忍失败的。就算是向老板说出你遇到了问题都可能会招致一顿数落。你的团队成员可能会因为你屡屡犯错而开始不尊重你。在很多公司中，即便是简单地承认自己犯了错误就可能导致严重的后果。[7]

这就是为什么很多团队选择忽视价值观和指导原则，或者只是嘴上说说却没有实际行动。基于以上原因，这种做法完全是理性的。假设把你自己放在一个文化与极限编程不兼容的团队中，如果你要求大家改变他们的工作方式，结果会怎样？你也许能够改变团队的文化，或者，你的团队成员和老板也许会对你非常不满而把你开掉。后一种远比前一种可能性大。当你听到别人说敏捷开发很难的时候，这肯定是原因之一。

注 7：敏捷新手常犯的一个错误是嘲笑那些不理解敏捷开发、没能成功应用敏捷开发的团队。这是不应该的。一项实践在某一个团队中起作用了，在另外一个团队中可能会导致整个团队被炒鱿鱼。对那些在敏捷开发的路上碰到麻烦的团队，应该试着多一些同情心。

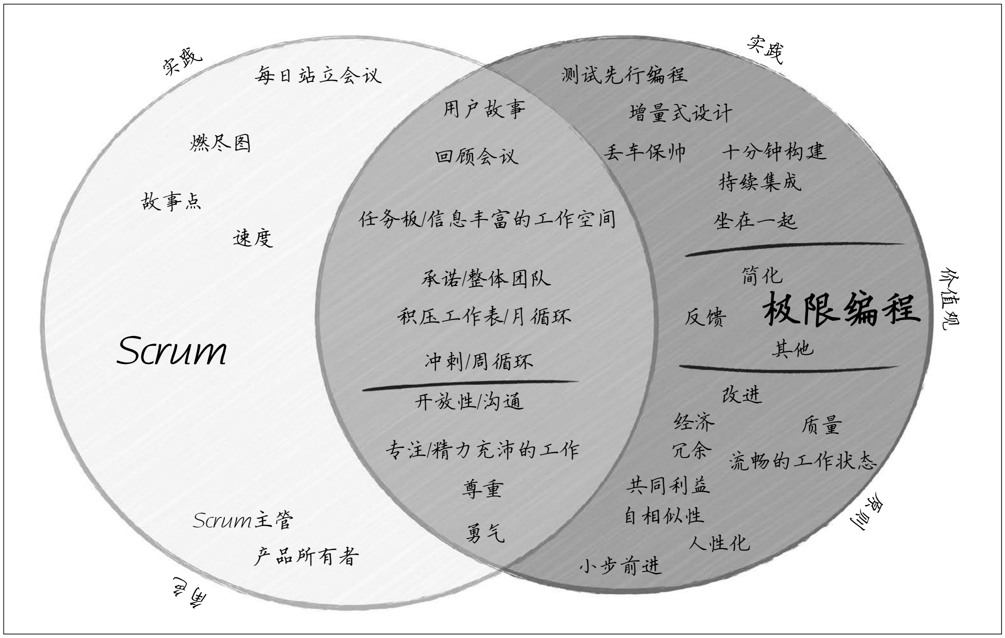


图 6-4：Scrum 与极限编程很类似，但是又很不同

6.7.2 极限编程指导原则可以加深对计划的理解

> 在一个成熟的极限编程团队中，角色和分工不是一成不变的。团队的目标是让每个人都能够发挥他最大的潜力来帮助团队取得成功。在开始阶段，固定的角色分工有助于学习新的习惯，比如让技术人员做技术方面的决定，业务人员则决定业务方面的事务。在团队成员之间那种相互尊重的新关系建立起来之后，固定的角色分工就会妨碍每个人发挥他的最大潜力。
>
> ——Kent Beck，《解析极限编程：拥抱变化（第 2 版）》

无论是 Scrum 还是极限编程，其价值观与团队文化间的冲突都导致了思维和心态层面的问题。这绝非巧合。Scrum 与极限编程的很多价值观和原则其实是相通的。不过这里面也有很多不同之处。比如角色分工，Scrum 有产品经理和 Scrum 主管这两个角色，但成熟的极限编程团队则没有固定的角色分工。学习极限编程的一个有效方法是专注于它与 Scrum 不同的那些指导原则。当学习极限编程团队是如何进行项目的计划时，这个方法尤其有用。

Scrum 的很大一部分是关于计划的：通过产品积压工作表来进行宏观上的规划；通过冲刺积压工作表进行一次迭代的计划；还有通过每日站立会议及其他 Scrum 实践来让团队中的每个人都参与到计划过程中。极限编程也使用类似的迭代方式：季度循环用来进行宏观上的规划；周循环用来管理具体的迭代开发；开发团队会通过类似任务板这样的跟踪工具来提升工作空间的信息量。

但是，计划和跟踪一个极限编程项目与计划和跟踪一个 Scrum 项目并不完全相同。指导原

则可以帮助我们理解这里的原因。为什么极限编程项目里没有具体的角色分工？这是因为极限编程团队非常注重*机会*和*多样性*。很少见到（当然，也不是完全没有）Scrum 团队中的产品所有者或者 Scrum 主管跑过来参与软件架构设计，也很少见到普通团队成员主导与用户的沟通或者对产品积压工作表的梳理。极限编程团队摒弃角色分工有两方面原因。第一个原因是，把某个人排除在一个角色之外意味着：当他或她有能力在这个方面做出贡献时，却因为自己的角色分工不同而错过了做出贡献的机会。第二个原因是，团队中的每个人都可能提供一个独特的视角，而这个视角也许就是解决某个棘手问题的关键所在。有时候，一个高度专注于技术的开发人员可能会对用户很了解，或者，一个项目经理可能就架构问题有些很有价值的见解，因为与那些从技术角度着眼的人相比，她观察问题的视角是完全不同的。

以这样一种方法来理解指导原则会让你对具体实践的执行产生一些变化。在那个虚拟篮球项目中，Danielle 发现，与一个对现有代码完全陌生的开发人员结对带来了一个完全不同的看问题的视角。这就是多样化在起作用：好点子可以来自任何人，哪怕是新人。这也是为什么很多极限编程团队会对结对编程的伙伴进行轮换，而不是一直与同一个人进行结对。这种轮换可以让结对变得更具*多样性*，于是也就意味着全新的眼光和全新的视角，这有助于发现更多的问题，也有助于激发创新。*人性化*这一原则也在起作用：让不同的人加入进来，让他们不断地在一起工作，有助于创造一个不断互相提供反馈的环境，这也会帮助每个人学会接受别人的反馈甚至批评，同时依然尊重提意见的人。同时，这也帮助那些团队里面资历尚浅的成员在发现问题的时候有勇气指出来，哪怕跟他们结对的是更加资深的前辈。多样性和人性化这种指导原则，加上沟通、尊重和勇气这些价值观，共同帮助一个团队使得结对编程成为一个更加有效的工具。

指导原则也能够帮助我们理解为什么极限编程没有一个类似 Scrum 回顾会议那样的事后检讨机制。极限编程非常重视*改进*和*反思*，而且极限编程团队不断地讨论他们现在做得怎么样以及如何才能做得更好。但是极限编程团队对过度的反省十分小心，反思过了头可能会对持续交付产品造成影响。因此，极限编程团队更倾向于把反思融入到工作中去。在这一点上，结对编程就很有效：让开发人员在写代码的同时讨论他们到底要干什么。*慢慢来*这一原则对此很有帮助：相对于设置一个大型的总体目标（“咱们一次性采用所有的极限编程实践吧”），开发团队可以一步一步慢慢来（“咱们先从结对编程开始，同时我们要确保每天我们都讨论要如何改进”）。

6.7.3 极限编程指导原则与实践相互促进

极限编程团队使用*故事*这个概念，如果你看一下他们的某个故事，你会发现跟 Scrum 团队所使用的故事没什么区别。但是，尽管这个实践在两种团队中几乎完全一样，你还是可以通过应用指导原则学到很多关于极限编程团队是如何使用这些故事的。

图 6-5 中是一个典型的极限编程团队可能会在项目中使用的故事。极限编程，与 Scrum 类似，并不会为用户故事规定一个统一的格式。在这个例子中，该团队决定使用树形的句式（而不是“作为一个……我想要……因此……”这种格式），同时把他们认为实现该故事所需要的工时数写在卡片的正面。

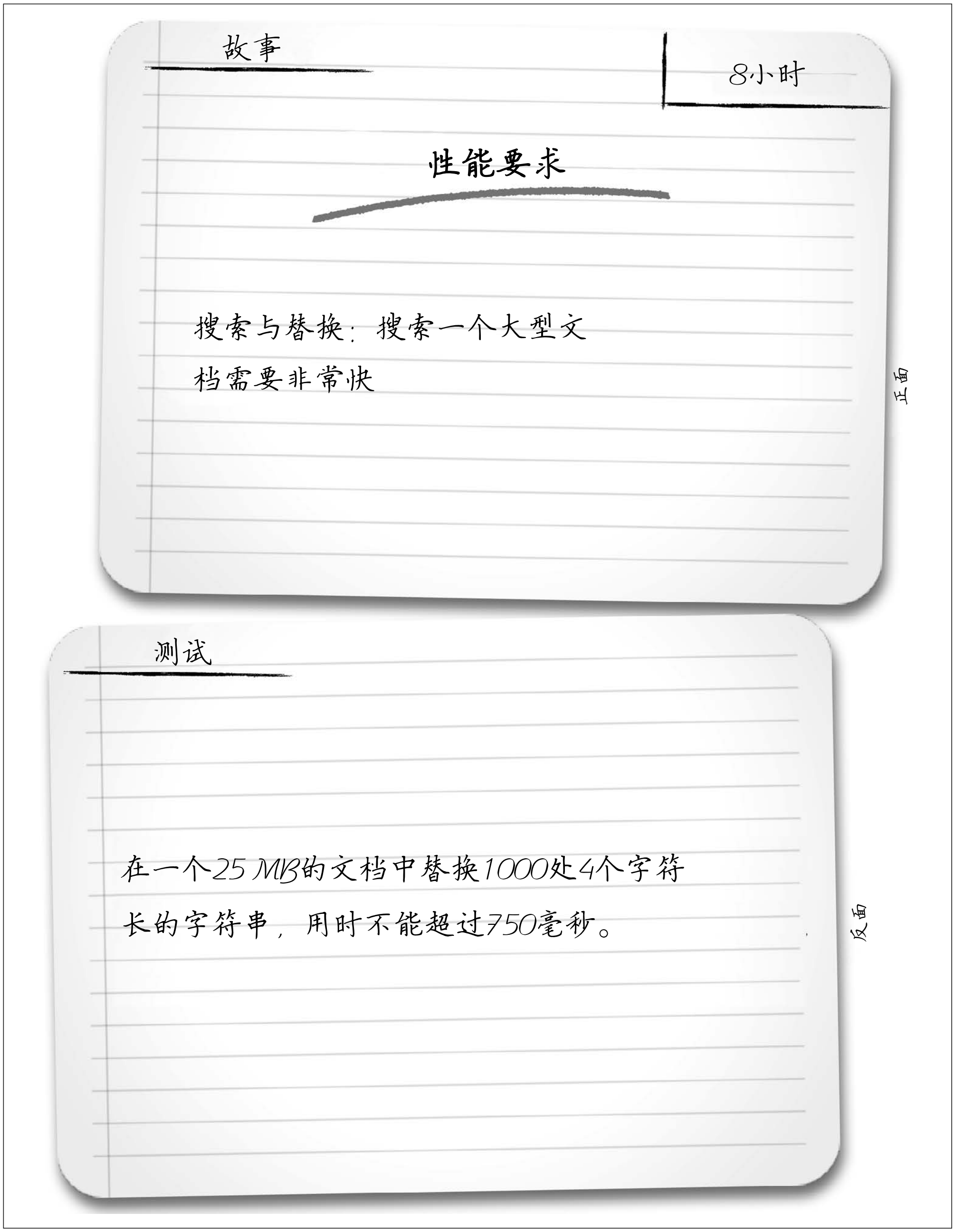


图 6-5：极限编程团队使用与 Scrum 团队类似的故事，但是会把他们自己的价值观与指导原则应用进来

你已经知道了一个 Scrum 团队会如何使用这个故事。它会被作为积压工作表的一部分，加入到某个冲刺中，然后贴在任务板上。那么，极限编程团队将如何使用他们的指导思想把

这个故事整合到项目中呢？他们可能也会做类似的事情。但是他们同时还会使用极限编程的指导思想来帮助他们理解该如何把该故事包含进项目中。

- *经济因素*
  由客户决定接下来要开发哪些故事。这样可以保证项目团队始终专注于那些最有价值的故事。
- *失败*
  故事都是比较小的，而且相对独立。开发团队可以完成一个故事并快速地把它交付给客户，这样如果做得不对，他们也可以及时更正。
- *责任明确*
  某个开发人员阅读一个故事卡并给出他的工时估计，那么他就是这个任务的责任人。
- *沟通*
  以用户的语言来撰写故事能够帮助他们安排工作的优先级。
- *质量*
  预先想清楚应该如何测试一个故事可以帮助你提高产品的质量。

因为你已经了解了用户故事是如何工作的，以及项目团队是如何使用它们的，你可以以此为起点，去理解上述的指导原则。

6.7.4 反馈循环

极限编程与 Scrum 另外一个相似之处在于，这两种开发方法都非常重视*反馈*。我们已经看到 Scrum 是如何通过反馈循环来保证开发团队与客户之间的持续沟通。Scrum 团队与极限编程团队在对*开放性*（Scrum）和*沟通*（极限编程）的强调上殊途同归。他们对于软件团队中的人们如何沟通有着十分相似的看法。

但是极限编程就反馈这一点来说有更深一层的考量。极限编程团队的迭代周期更短（每周一个循环），以此来增加反馈的次数并缩短整个反馈循环。在 Scrum 中我们学习了如何让迭代更加高效，团队需要积极地去理解用户的需求以及什么东西对他们最有价值。为了让反馈循环更短，他们持续地反思项目的进展。一旦发现问题，团队就会展开讨论。因坐在一起而产生的渗透式沟通有助于把相关的反馈信息传递给整个团队。把这些都结合起来，开发就可以达到*流畅*。这项指导思想所关注的就是要找到一条直接而高效的道路，使得开发团队能够通过它持续地交付高可用性、高质量的软件产品。

你已经看到了极限编程的各个不同的组成部分（坐在一起、渗透式沟通、反馈循环、沟通、反思）是如何结合在一起帮助团队找到一条高效、高产的道路的。但是要想做到流畅，有一个必要条件是诸多组成部分都要能够协同工作：实践、指导原则、想法，还有最重要的，优秀的软件设计与编码。一旦做到了流畅，整个团队都会感觉到他们在不间断地交付软件。由于更多的沟通、获取更多的不间断的反馈，他们开始意识到软件开发方法中存在的瓶颈，同时也学会了如何作为一个团队一起去克服遇到的障碍。没有了瓶颈，一切更流畅了。



图 6-6：当团队具备了正确的极限编程思维时，它的实践就更感觉是软件开发的最有效方法了

对于开发团队来说，这些不是一朝一夕就能够做到的。他们不仅要花时间去开发软件，还要讨论他们是如何做开发的，以及学习每一项指导原则和实践如何能够帮助他们进一步地提高和完善。在这里，“慢慢来”这一原则又派上用场了，它能够帮助团队设置合理的目标，比如缩短反馈循环以便更容易做到流畅。对于整个极限编程方法的采用可以被看作是一个路线图。这个星期他们可能会想办法增进反思，或者通过坐在一起来改善工作环境，或者更有效地结对编程。只要整个团队在向着正确的方向前进，他们就清楚地知道他们正在改善项目的计划、跟踪和执行等。而这意味着他们每天都在进步，不仅仅是在极限编程方面，在软件开发方面也是如此。



要点回顾

- 极限编程的 5 个价值观可以帮助团队进入正确的思维方式，避免“聊胜于无”的结果。
- 极限编程的**沟通**价值观是指每个团队成员都应该了解他或她的同事正在做什么，并且随时与他们保持沟通。
- 极限编程团队重视**简化**，应该尽量使用简单直接的方案，避免复杂的方案。
- 通过持续的测试与集成，极限编程团队创造了一个**反馈循环**以保持他们的高质量。
- 身处极限编程团队中的人们拥有作出最佳选择的**勇气**，哪怕是作出像“完全抛弃之前的糟糕方案”或者“为了让代码将来更容易维护而加班”这样的选择。
- 极限编程团队中的每个人都**尊重**他或她的同事以及他们的经理和用户。

常见问题

那么我应该从哪做起呢？

这个问题经常被刚接触极限编程的团队问起，而且对于这个问题没有一个标准答案。这取决于你所面临的问题是什么。翻到本章最前面，看看那一条关于人们有多么痛恨变化的引言。不要低估这种痛恨。

采用极限编程意味着你需要改变自己开发软件的方式。一旦团队中的人意识到他们要以与以往全然不同的方式去工作，那种对新实践的兴奋感觉很快就会消失不见。

那么你应该从哪里做起呢？要是你在考虑采用极限编程的解决方案，那么你可能有一些亟待解决的问题。这些问题是什么？哪些极限编程的实践有助于解决这些问题？

如果你的问题是 bug 太多，那么先试试测试驱动开发。如果你的团队缺乏沟通或者大家总是意外地收到需求发生变动的消息，那就先试试使用故事。不过注意，先尝试一项实践，就一项，而且注意不要只做表面文章。与整个团队一起坐下来，一起讨论极限编程的价值观和指导原则。确保每个人都理解你采用这项实践是为了达成什么目的。如果每个人都能理解你要解决什么问题，以及这对他们有什么直接的影响，同时你可以评估该项实践到底起没起到作用、起了多大作用。这种评估有助于你在必要时对实践进行调整，但是，更重要的是，它帮助每个人了解该项实践是怎样改善了他们的生活。这可以消除他们那种浪费了宝贵时间的感觉（这种感觉也是导致实践被慢慢遗忘的原因）。

对于结对编程，我有点头疼，我看不到我的项目中有人这么做。这是为什么？

当一个团队尝试采用极限编程但最终还是回到他们之前的老路上的时候，结对编程往往是最先被抛弃的实践。之所以这样是因为在结对编程这一点上，团队的旧观念与极限编程的价值观的冲突最为明显。因此，如果结对编程在你的团队里不太现实，那就想一想每一个具体的极限编程价值观。针对结对编程，你可以玩一个游戏，问自己“使用下面这项实践还好吗？”

沟通

你能否做到在编码的同时还跟你的同事说话？你的同事可以跟你说话吗？你不在的时候大家讨论你的代码呢？你能接受吗？

简化

别人指出你刚写的那部分代码太复杂了，你能接受吗？你愿意马上花时间去修正它吗？写一行“待完成”的注释就算了？如果马上就要到截止时间了呢？你会怎么做？

反馈

你能否接受别人说你的代码写得不好，而且可能是不客气地说，因为程序员可不怎么会掌握分寸？

勇气

你能否做到在看别人编程的过程中指出问题的所在，哪怕这有可能导致一场争论？如果你知道你可能说的不对呢，你还会这么做吗？你能接受自己犯错吗？你能接受自己在另外一个开发人员的面前犯错，并被他或她注意到吗？如果那个人是你很尊敬的人、或者比你资深的人呢？

尊重

如果有人说你的代码写得不好，你是否愿意倾听，并且意识到他们有可能是对的、而你有可能是错的？你能做到不记仇吗？

如果你和你的团队不能完全做到上述这些，那么你多少也就知道该从何处下手了。哪个问题最让大家感觉到不舒服或者无法接受？就该问题展开讨论，看看到底大家为什么对它感到不适。给你自己和你的团队设置一些目标，看看你们怎么才能慢慢地把这种不适逐步消除掉。

或者，在最坏的情况下，如果说思维方式上的问题根本就不允许你和你的团队采用结对编程这种形式，那么找一个你们觉得 OK 的实践，先做起来。不过不要形式化地做，一定要保证你的整个团队充分地讨论极限编程的价值观，并且真正把那些指导原则当回事。过一阵子，很可能你再回头尝试结对编程时就会发现你的思维方式发生了转变，因为你已经在其他实践上做了一些小幅度的尝试了。

我的程序员为什么需要去写那些测试？我的团队没有那么多的人手在那干坐着专门负责挑 bug。这难道不是更便宜的 QA 人员应该做的事情吗？

测试先行编程远远不是写测试和运行测试那么简单。你是不是对测试先行编程或者测试驱动编程感到不适？这说明你需要仔细研究一下极限编程的价值观和指导原则。很可能你对它们当中的某一条也不太认同。你也许不太认同有关质量的那条原则，因为这条原则指出，团队中的每个人对项目的质量都负有责任。你不能简单地把代码扔给一个测试人员，然后让那个人负责找出所有的 bug。或者，你也许不怎么认同那条关于冗余的原则，因为那些测试似乎给代码仓库引入了额外的、有可能重复的代码。花上几分钟时间，仔细研究一下极限编程的指导原则和价值观，尝试分析一下，你究竟不认同哪些东西。

确实，编写单元测试会占用开发人员的时间，而且对于没写过很多单元测试的程序员来说，常常会感觉这是对他们时间的一种浪费，或者这种低级工作不应该是他们做的。[8] 可是一旦程序员开始使用测试驱动的开发方法，总会有那么一天，一个单元测试没能通过，而这个失败的测试会揪出一个原本要花费好几个小时才能发现的难缠 bug。这样的事情发生上几次，一个原本对测试驱动充满怀疑的开发人员将变成它的真正信徒。很多敏捷开发人员都这样说，发生这样的事情会让你变得“被测试感染了”，而测试驱动开发也变成了软件开发的显而易见的方法。

还有一件事需要注意：测试先行编程的内涵远比表面看起来的要多。在第 7 章中，你将学习到先编写单元测试对于软件的设计有什么样的深远影响。一个有效使用测试驱动开

注 8：这些人其实是在说：“我没时间去找出我自己代码中的 bug，我正忙着制造 bug 呢！”

发的团队能编写出更好的软件，不仅仅因为他们能够揪出 bug，更因为他们的开发方法与不使用测试先行的时候大不相同。

还是没有被说服吗？没关系。你可能只是需要从一个不同的角度来接近极限编程。幸运的是，还有很多其他实践可供你选择。就像结对编程那样，如果你不认为测试先行编程是正确的做法，那么你可以先试试别的实践。

我是一个程序员，你说的这些让人感觉有点太松散随意了。我们通常都是通过票证分配工作任务，这样每个人需要做什么就很明确。可是在极限编程环境中，我怎么才能知道接下来应该做什么呢？

如果你习惯于使用项目跟踪系统或者甘特图等工具进行项目跟踪，那么极限编程对于每天的项目工作分配可能在你看来十分不同。就你的情况来讲，每天你会得到一个清单，上面有你需要完成的各项任务，随着你把清单上的条目逐个勾掉，你就知道你正在完成自己的工作。这种工作方式挺好的，因为你总是清楚地知道你在取得进展。这对你的老板和上级也是不错的方式，因为他们可以很容易地了解项目的状况。

一个习惯于上述工作方式的开发人员开始可能会觉得一个成熟的极限编程团队很难合作。这是因为在一个上令下行的环境中，已经有人帮你获取了需求信息并把他们拆散成一个个独立的可供你执行的步骤，任务分解结构的条目、任务票证等。这给了所有人一个很清晰的结构框架，因为所有这些决定都已经在项目开始的时候做过了。

极限编程团队不会预先制订所有的计划。作为季度循环的一部分，他们会讨论主题和故事，但是他们的迭代周期非常短，只有一个星期。单个的任务只会以“周”为周期进行计划。这也算是极限编程比较“极限”的一面吧，这绝对是一种比较极端的、在最后时刻才做决策的方式。而对于习惯了很早就把这些决策制订好的开发人员来讲，这种极端的方式似乎有点把自己逼得太紧，几乎没有余地了。

另外还有一件事，让习惯于传统项目管理方式的程序员感觉极限编程有点杂乱无章：结对的伙伴不断地更换，而且大家都不会提前去计划谁应该做哪一项任务。某一项工作具体由哪个人来做，直到马上开始要做这个任务的时候才真正定下来。在这一点上，极限编程与 Scrum 是不一样的，因为 Scrum 会在每日站立会议上审查冲刺计划，并通过自组织进行任务分配。由于极限编程团队的迭代周期特别短，其反馈循环的周期也很短，所以他们可以把任务就堆在那里，大家可以随时从这一堆任务里拿出一个来做。

像你说的那样，如果一个团队里的人就随随便便地从一堆任务里随便拿一个出来做，这工作能干好吗？难道说不应该预先计划好具体谁应该负责哪项工作吗？这才能做到让那些具备相应专门知识的人去做合适的工作啊。

不是这样的。事实上，这正是极限编程的另外一个亮点，它能够帮助团队提高和进化。在周循环开始的时候，开发团队会分析他们要实现的那些故事，把它们分解成具体的任务。因为开发是迭代式的，在一个迭代周期结束的时候他们会交付“真正完成了的”可用软件，而所有完成了一半的故事则归入下一个周期。但是他们不会预先计划哪个任务由谁来做，而是把任务堆在一起，或者放在一个队列里，或者采用什么其他形式的集合。当某一对程序员完成了手头的工作时，他们可以拿出队列里面的下一项任务开始做。

那么这是不是说当你去拿下一个任务时，即使该任务有更合适的人可以做，你也必须把它拿来做呢？回答当然是否定的。团队的成员并非机器人。他们会以他们能力范围内所能够达到的最佳状态去完成工作，在这个过程中作出尽可能合理的决定。而这些决定，可以在开始做某一项任务之前才做。对于专门知识这一点，可能轮换结对编程伙伴也会有些帮助，因为通过轮换，具备某项专门知识的人就可以与另外一个希望学习该方面知识的人结对，这样，等下一次有某个任务需要这项专门知识的时候，就有两个人可以胜任了。

这也是在极限编程中没有固定角色分工的一个原因。关于这一点，Kent Beck 在《解析极限编程：拥抱变化（第 2 版）》中的描述如下所示。

> 团队成员之间建立起相互尊重的新关系之后，固定的角色分工就会妨碍每个人发挥他的最大潜力。如果一个程序员是撰写一个故事最合适的人，那这个故事就应该由他来撰写；如果项目经理最适合提出有关架构优化方面的建议，那就应该由他们来提出这种建议。

这里面反映出以下两个重要的极限编程原则。

责任明确

一旦某一对程序员承接了一项任务，他们就应该全力投入把它完成。如果他们遇到问题，就应该全力克服。但是他们也会向团队中的其他人寻求帮助，即使这么做可能会有点挫伤他们的自尊。（由于大家彼此坐得都很近，最好是其他人无意中听到他们遇到了麻烦，从而主动提供帮助。）

机会

每个新任务都是一个学习新东西的机会。如果某项技术有人不懂，学习该技术就成为了任务的一部分。这有助于知识的分享，而且给未来提供了更多的机会。

这里还涉及另外一个概念：共同所有权。在前面提到的 13 项极限编程的主要实践之外，还有以下 11 项衍生实践。

真正的客户参与

让客户参与到每季度和每周的计划会议中，并且真心倾听。

增量式部署

对系统的每一个小部分进行单独部署，而不是做大规模的一次性部署（同时，相信这种增量式部署方法是可行的）。

团队的连续性

让那些高效率的团队始终在一起工作，不要拆散他们。

缩小团队规模

随着团队的不断进步，他们完成工作的速度会越来越快；这个时候不要给他们增加工作量，而应该减少一个团队成员（让这个人把极限编程的文化带到其他团队中去）。

根本原因分析

出现问题的时候，找出问题所在，然后分析问题之所以产生的原因，并从根本上解决该问题。

共享代码

开发团队应该只维护生产代码和测试代码；文档应该从代码库中自动生成，而项目历史则通过口口相传的方式予以保存（因为即便有书面的项目计划，也很少有人去查阅）。

唯一的代码仓库

不要维护好几个不同版本的代码仓库。

每天部署

每天都将软件的一个新版本发布出去。

就合同范围进行谈判

作为一个咨询公司，不要把合同范围定死了，然后去就工期讨价还价（这样通常会导致为了按时完工而牺牲产品质量），相反，把工期先敲定，然后随着项目的推进逐步协商项目的范围。

按使用付费

这也是一个与咨询相关的实践，不要按开发工作量收费，按客户对系统的使用进行收费；这样你会得到实时的、不间断的反馈，从而了解用户使用了哪些功能，哪些功能没有用到。

我们在本书中不会太多讨论上述这些衍生实践，但这其中有一项有必要一提，那就是分享代码。在一个极限编程团队中，跟项目相关的每个人（包括项目经理，只要他或她是团队的一份子）都应该有权对代码的任何部分进行编辑修改。每个人都是所有代码的主人，大家都有一种主人翁意识。这意味着如果一个团队成员发现了一个 bug，他就会修复它，即便这个 bug 是别人引入进来的。这跟很多传统团队的做法很不一样，在那些团队中，每个人只对他或她自己的那部分代码负责。

（我们还会深入讨论另外一项衍生实践："根本原因分析"，以及"五个为什么"技巧，这会帮助你在遇到问题时透过表面现象从根本上解决问题。我们会在第 8 章中介绍这些内容。）

打破所有权的藩篱，有助于让整个团队团结起来，因为当出现问题时，大家都感觉到自己负有解决该问题的责任。因此，如果每个人都对代码有一种责任感，那么每个人也就一样有能力解决队列中的每一项任务，或者至少有能力学习如何解决每一项任务。大家会把学习的过程视为项目的一部分，值得他们花时间去做。

真有 11 项衍生实践啊？极限编程的实践怎么这么多呢？

如果你感觉极限编程的实践太多，有点接受不过来，那么这说明你可能就是那种一心想给极限编程实践列清单的人。我们之前已经看到了，这种思维只会给你带来"聊胜于无"的结果。衍生实践存在的目的是帮助开发团队解决他们在成长过程中遇到的问题。

如果你脑子里在想“我要怎么才能一下子把这些实践都做到呢？”，那么你就应该注意了，你很可能有清单情结。还记得关于“慢慢来”的指导原则吗？如果你肯花时间去理解每一项实践对你和你的团队有什么作用，那么把它们整合到你的团队中并逐步完善就应该不是什么难事了。

在第 7 章中，你将学习极限编程的诸多实践是如何共同作用并对开发团队如何设计软件产生重大影响的。当你读到那一章的时候，尝试着找一找这些实践是如何共同作用并互相促进的。这将有助于你摆脱清单情结，转而从一个更加全面的角度看待极限编程。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 给结对编程一个机会，哪怕一开始感觉有点奇怪。你不必非得永远坚持下去；就尝试几个小时，看看感觉如何。你可能会发现它没你想象得那么难。
- 如果你是一名开发人员，尝试一下持续集成。你不必非得让整个团队一起尝试（暂时不用），你自己今天就可以试试。下次编码间歇的时候，把最新版本的代码签出到你的沙盒中，进行测试，直到你觉得差不多集成完毕了。隔几个小时再做一次。有没有遇到现在很容易解决、但留待以后却会让你头疼不已的问题？
- 尝试一下测试驱动开发。下次你创建新的类（或者模块、子过程或者你所使用的语言中的任意单元）的时候，先写一个单元测试。不必非得写出考虑了每一种可能的边界情况的完整测试；就写一个简单的、可运行的、但是现在无法通过的测试即可。你可能得学习一下在你的编程语言中如何使用单元测试；当然，这也是一个很不错的练习。

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 《解析极限编程：拥抱变化（第 2 版）》：深入了解极限编程的实践、价值观和指导原则。
- 《单元测试之道》（*Pragmatic Unit Testing*）：深入了解单元测试和测试驱动开发。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 这是敏捷开发的教授中最具挑战的部分，如果教练不具备编程背景，则更是如此。把重点放在极限编程价值观和指导原则上，帮助团队发现这些价值观和原则是如何影响代码的。
- 有时候团队不容易理解勇气和尊重这两条原则。帮他们探寻团队犹豫不决，不知道是否应当让外界知道代码实际质量的情景。比如，他们会不会对 demo 做些手脚从而绕过某个已知的 bug ？或者修改一个 bug 报告，使它看起来没那么严重？帮他们找到就这些问题进行沟通的更好办法。
- 沟通还可以怎样进行改进？帮助他们寻找机会，制造信息辐射体。

第 7 章

极限编程、简化和增量式设计

我算不上卓越的程序员，我只是一个有着卓越习惯的不错的程序员。

——Kent Beck，极限编程创始人

极限编程的目标不只是让团队更好地工作。极限编程及其实践、价值观和指导原则的更大目标是帮助团队开发出易于扩展和变更的软件，同时帮助团队成员在一个接受和拥抱变化的环境中一起工作、计划和成长。

采用极限编程不仅仅意味着要通过结对编程来做不间断的代码审查，或者要通过测试驱动开发来增加测试的覆盖率。更高的产品质量以及与同事和谐共处也是极限编程的副产品，而且这些副产品对于实现主要目标也是十分有帮助的。上面所说的这些都很好，而且它们能够改变你开发软件的方法，但是它们并未从根本上改变软件的设计。

值得反复强调的一点是：极限编程的一个重要目标是使得软件可以很容易地进行修改。如果软件修改起来很容易，那么开发团队就会更愿意拥抱变化。这一点对于优秀的极限编程团队如何设计软件、如何编写代码有着深远的影响。

进入正确的极限编程思维还有非常关键的一点：要真正相信你在第 6 章中学到的那些实践（比如测试驱动开发、结对编程以及“丢车保帅”）能够帮助你和你的团队从一个全新的角度思考软件设计，而不使用这些实践则会导致你编写出质量不佳、难以修改的代码。

学习完本章之后，你将明白，即便是聪明的程序员也可能写出存在着严重的设计和编码缺陷的代码。在本章中还将学习极限编程的最后三项主要实践，以及它们如何帮助你避免那些严重的设计和编码问题。另外，我们还会介绍很多极限编程团队成员养成的好习惯（就是 Kent Beck 在本章开头的那句话中所提到的那些习惯），以及极限编程的各项实践是如何形成一个生态系统，并使你能够写出更好、更易于维护、更灵活而且更加容易修改的代码。

故事：有一个正在开发虚拟篮球网站的团队

- Justin——一位开发人员
- Danielle——另一位开发人员
- Bridget——团队的项目经理

7.1　第4幕：再次加班

Justin 一直希望能按时下班，赶上傍晚 5 点 42 分的地铁。但是不知怎么的，他很少按时下班。今天也不例外。而且每次加班的起因似乎一样：可能是有个小 bug 需要修复，或者需要对代码做个小调整，这个 bug 或者调整通常出现在他下班前一个小时。不知不觉地，这个小修改就变成一个不可收拾的怪兽了。

几乎一直都是这样一个模式。一个看似很简单的修改，改到一半，Justin 就会发现他需要修改代码的另外一个部分。行，没问题，他把那部分也改掉就是了。可是要完成这第二处修改，就需要同时修改第三处、第四处代码。其中的某一个修改可能又需要他去修改另外一部分代码，如此恶性循环。有时候，当他把需要修改的地方全部找出来时，他都忘了自己最开始为什么要做这些修改了。

有时候，要做的改动实在太大，他没法下手。Justin 会跟 Danielle 和 Bridget 商量，最后得出的结论往往是：修改风险太大，可能导致产品不稳定。特别让人受挫的是，你花了大量的时间去做一个修改，但最后却不得不放弃全部的工作成果。

今天晚上还是老样子。Justin 还在重复着“一个修改连着一个修改”的模式。Justin 还记得他曾经告诉 Danielle，“我答应了我女朋友早点回家，所以我做完这个就下班。”而他要做的事情不过是在设置页面的一个下拉框里增加一个新选项。很简单的。

这是 5 个小时之前的事情……

Justin 要是早知道需要花这么长时间，他可能就回家再做了。或者干脆今天不做了，留到明天再说。

Justin 要修复的是一个包含系统中虚拟球队的球员配置统计信息的下拉菜单。他只不过想给下拉菜单增加一个选项，从而允许用户隐藏球员的排名信息。他把这个选项加上了，但是只要他在下拉菜单中选中该选项，页面上就会弹出“你必须从列表中选择一项”的消息。为了解决这个看似简单的问题，他已经花了好几个小时了。

当 Justin 对页面的验证代码进行调试时，他发现数据的来源是某个数据表的缓存副本，所以，他就在用户单击“确定”按钮之后增加了刷新缓存的调用，而这又需要他修改其他三个地方的代码。然后，有三个其他的页面也要复用同样的设置页面，他又不得不去对那三个页面也进行了修改。

全部加起来，Justin 差不多对代码做了十几处改动，不过最后他让系统能够正常工作了，

多亏了一个差劲又山寨[1]的解决方案。他征求了 Danielle 的意见，她想出了一个解决问题的办法。“刷新缓存那里只用到了用户的 ID。你可以试试创建一个伪用户类，只给它设置一个 ID，其他的属性一律返回空值。”这个办法很丑陋，但确实管用。

Justin 终于把那个额外的选项加到下拉框里了。那种熟悉的加班感觉老早就已经袭来了。他对 Danielle 说：“终于弄完了。我得回家了。为了加这么一个小破功能，我在多个部分之间来回折腾，唉，我算是知道被踢来踢去的皮球是什么感觉了。”

Danielle 说：“完全理解。我这也是，本来想改一下登录页面，但是发现要做这个改动还需要修改其他三个地方。其中两个修改又涉及修改其他好几个类，而其中有一个类调用了某个服务，我完全不知道应该怎么用那个服务给过来的响应。”

Justin 问：“为什么总是这样呢？”

Danielle 说：“编程就是这样吧。问题是，你真的确定你刚写的代码是正确的吗？”

话一出口 Danielle 就后悔了，因为她自己也不确定了。她和 Justin 两个人面面相觑了半天。“我看我还是再花点时间测试一下吧……”Justin 把耳机带上，然后再一次开始给他的女朋友发道歉短信。

7.2 代码和设计

> 在 C3 项目中，Kent Beck 需要把团队文化中对“聪明代码”的崇尚转向对简单解决方案的追求。这是一个众所周知的难题。他的其中一个技巧是利用来自同事的压力。他会组织一些特定的仪式，比如：让大家围成一圈，给那个写出聪明过头代码的人戴上一顶帽子，帽子上有个螺旋桨。然后，开动螺旋桨，并让大家就这个人的“小聪明”发表看法。因为不愿在同事面前出丑，所以大家都会远离“聪明的”解决方案；对简单设计的赞赏让大家更倾向于采用简单的解决方案。当然，人与人是不同的，团队中并不是每个人都能够接受极限编程的做法。有一个人不习惯这种要求遵守约定、紧密协作的新工作风格，所以最后离开了该项目。
>
> ——Alistair Cockburn，《敏捷软件开发（原书第 2 版）》

极限编程团队写出的代码很容易修改。这是如何做到的呢？谁也不希望写出难以修改的代码啊。

事实上，很多开发人员（甚至一些非常聪明的开发人员）所做的正好相反。他们口口声声说要构建可以复用的代码，然后花大量时间去设计他们心目中完美的、高度可复用的模块。但是，你很难为将来未知的情形预先做好设计，反而很容易把代码搞得过于抽象和宽泛，结果造成用来构建框架的代码甚至与真正用来实现业务逻辑的代码量差不多了。说来让人吃惊，今天为了实现代码复用而做的聪明设计，可能明天就变成大家前进道路上不得不绕着走的绊脚石，碰都不敢碰。

很多人以为那些容易出问题却不容易修改的代码通常是新手写出来的。但现实中更常见的

注 1：这里的“山寨”指的是那种丑陋、粗糙、不够优雅且难于维护的解决方案。

是，很多最优秀的开发人员也会写出这种难以修改的代码。（缺乏经验的程序员很少有机会写出足够复杂且庞大的代码）这倒不是因为他们开发的软件就一定写得不好或者设计得不好，更多的时候，这是一个"聪明"与"简单"的权衡问题。

对开发人员来说（尤其是优秀的开发人员），在编码时不仅要考虑到眼下的问题，还会自然而然地考虑将来可能产生的问题。相信很多人都有这样的经历，计划会议开起来没完没了，因为无论设计多么周密合理，总有人能够想出一个难以实现的边界情况。越是重大的问题，越是要花费更多的时间来解决。

Scrum 团队把整个项目分解成多个冲刺来避免无休止计划的问题。Scrum 团队只专注于今天的问题，剩下的则留给将来的某个冲刺解决。Scrum 团队把决定留到最后责任时刻再做。而极限编程团队的季度循环和周循环达到的是相同的效果。

把决定留到最后责任时刻不仅仅是一项用来做计划的工具。它对设计、架构和编码也很有价值。它是极限编程团队做到简化原则的主要手段。

如果你以开发人员的身份在几个不同的团队中工作过，那么很可能亲眼见过难以修改的代码是什么样子。当初写代码的人也不想这样。当初写下这些代码是为了实现某种功能，只不过现在需要实现的功能发生了变化。经过几轮这样的需求变更，代码就变得剪不断、理还乱了。那么，为什么会发生这样的情况呢？

7.2.1 代码异味和反模式（如何判断你是不是聪明过头了）

简化，从知道什么时候不该做得太多开始。

我们来举一个例子，说明复杂化（简化的对立面）如何影响一个产品。本书的作者之一上大学的时候曾经收到过一份礼物。礼物是一个集榨汁机、搅拌机、食品加工器于一体的厨房电器。但在这么多功能中，哪一项都算不上优秀：无论是还不错的榨汁功能、很一般的搅拌功能，还是完全没法用的食品加工功能。它还有一些在这类电器上很少遇到的其他问题。比如，它的零件形状很怪异，很难存放，而且几乎没法清洗。

不管怎么说，大学毕业十年后，我们故事的主人公并没有购买一个新的搅拌机或者食品加工器，因为那样似乎有点多余。但是，因为那个"全能王"电器的食品加工器实在太差劲了，所以他做菜的时候就专挑那些不必用到食品加工器的菜来做。当他终于决定把这个劳什子束之高阁（他实在舍不得把一个"好好的"电器扔掉）并从附近的连锁药店买了一个最便宜的食品加工器的时候，他突然发现，现在他可以做那么多他原来没法做的菜肴。而他自己则纳闷为什么花了 10 年才走出这一步。

上面提到的那台电器还不如没有。因为它没让本书作者吃上那些需要用到食品加工器的菜品，它简直是妨碍他去做那些菜。与此同时，因为它的存在，又让"再买一个食品加工器"这个选项看起来没有必要，所以，多年来，本书作者一直避免去做那些需要食品加工器的菜品。并不是他不会做，而是因为用那个劳什子做菜实在是太费劲了，划不来。但这么多年来他都没想过要买个新的，因为他已经有一个"好好的"电器了。要是他手头上没有这个家伙，他肯定会买一个。

这个电器之所以糟糕就是因为它太复杂。要同时具备三种不同的功能，就需要额外的零

件，还需要在技术上作出一些妥协和让步，这些都会导致最终的设计违反直觉，难以使用（比如，你很难把搅拌机连上，因为如果位置没有完全对准，它就不转）。要是一开始就用三个简单的产品（比如说，一台便宜的榨汁机、一台便宜的搅拌机和一台便宜的食品加工器）来替代这个“全能王”，那么花费会更少、存储起来也更省地方，在厨房里用起来也更灵活。

这就是为什么极限编程的价值观中包含“简化”这一条。极限编程团队很清楚，正是这种复杂性设计会给项目的计划带来很大影响，正如我们在前文中看到我们的 Scrum 团队所遇到的那些麻烦一样。

相信每个人都会同意，要使一个团队保持高产，不应该有人闲着没事干。但是，有些上令下行式的经理在这一点上做得有点过头了：他们会制订一个 100% 占用所有资源的计划[2]，并且要求每个开发人员记录其在每个开发任务上所花费的每一分钟。这种为了达到 100% 资源利用率、为了保证每个人的每一分钟都在干活的做法会给项目管理带来很多额外开销：每个人都要记录他们的时间花在哪里，而且要把这个信息录入跟踪系统（这听起来可能没什么大不了，但是实践起来可能很费时间，同时也很费脑力）；经理则需要制订、更新以及审阅计划；还有大量无法避免的会议，讨论“解决”某个开发人员只有 95% 的时间在干活这种问题。

这是一种非常复杂的项目管理机制，更重要的是，这些额外的复杂性并不能够为项目创造价值。大家需要花费大概 20% 的时间来记录和汇报他们在另外 80% 的时间里做了什么；这就导致整个项目要花费更长的时间才能完成。开发团队好像将越来越多的时间花在进度通报会上，花在回答“你的时间利用率是百分之几”和“能否估计一下这个任务还需要多少分钟能够完成”这类问题上。

极限编程团队使用迭代方式，把项目计划的决策推迟到最后责任时刻，就像 Scrum 团队所做的那样。这样一来，对项目进行计划的复杂度就大大降低了。极限编程团队通常也不会像上面例子中那样随时监控每个人的时间，因为这种监控并不会对完成项目有所助益。虽然出发点是好的，通过监控获得的信息却很少被用来做下一步的规划，而且在项目结束后基本上就无人问津了。更糟糕的是，我们前文已经提到，项目是不断随时间变化的。为了达到 100% 的资源利用率，整个团队就需要把所有项目相关的决定在最一开始的时候都定下来；一旦发现某一项决定是错误的（总会有的，这是项目开发的自然组成部分），就需要整个团队重新考虑其计划、任务分配和资源利用率。

在这个例子中，我们还能学到一件事情，就是要尝试发现模式并用它们来改进你的工作方式。

如果你曾经在这样的团队中工作过：项目经理采用的是那种严格上令下行式的管理方式，要求每个人都用尽 100% 的时间，不断召集会议检查每个人的进度以保证时间都花在了工作上，同时要求每个人不断更新他们的任务以达到那个不切实际的要求……那么，当读到前面几段的时候你可能会感到不安和愤怒，而且可能还会勾起你对无休止进度报告会的痛苦回忆。这是因为你刚刚看到就是一个*反模式*：即会给项目带来麻烦和问题的行为模式。

注 2：这种上令下行式的项目经理总爱把团队成员称作“资源”，哪怕不是正式场合，也是如此。

发现项目管理中的反模式可以让你发现问题，同时有可能帮助你找到简化项目管理方式的途径，从而让团队能够重新专注于产品的开发。

7.2.2 极限编程团队主动寻找和修复代码异味

代码当中也存在反模式，就像项目管理中的反模式一样，识别代码中的反模式是消除代码复杂性的第一步。当某个反模式直接与代码的架构或设计相关联的时候，我们就把它称为*代码异味*（code smell）。极限编程团队会不断寻找这些代码中的异味，并养成习惯，一旦发现就马上修复它们，而不是任它们继续恶化下去。

我们会花些时间讨论几种最常见的代码异味，这样你就能够更好地理解一个高效的极限编程开发人员的思维方式。这个列表并不是随意列出的，它是经过众多开发人员从他们多年的项目经验里面总结出来的。"代码异味"和"反模式"这两个术语是由 Ward Cunningham（也是敏捷宣言的作者之一）首次提出的，并在 20 世纪 90 年代晚期在开发人员当中流行起来的［这还要归功于 WikiWikiWeb（http://www.c2.com/cgi/wiki?CodeSmell），它是第一个 Wiki 网站］。WikiWikiWeb 曾经是（在某些圈子里现在也还是）最受开发人员欢迎的讨论软件设计的地方之一。越是更多的人加入到讨论软件设计和架构问题的行列中来，大家就越发现他们所遇到的问题有很多共同的表现。[3]

很多软件工程师（包括本书的两位作者）初次遇到这些问题的时候都会有种发自内心的激烈反应。这是一种很奇怪的感觉：让你纠结、无计可施的问题被一个陌生人完美、精确地描述了出来。而这其实是需要我们在接下来的讨论中时刻铭记的一点：这些问题是人制造出来的，又是人识别出来的，并且可以由人来防止或修正。问题的表现可能是技术层面的，但是其根本原因则影响到团队中的人，因此，解决方案一定是技术和团队两方面兼顾的。

当我们培训开发人员时，有一种代码异味，绝大多数人都会点头表示见过。我们把它称为*枪伤手术*（shotgun surgery）。这种代码异味是指：当你尝试对代码的某个部分做一个小修改时，却发现需要同时修改另外好几处似乎完全不相关或勉强有那么一点关联的代码；你尝试对它们进行修改，结果发现其中的一个地方又需要修改其他地方，而且又是显然不相关的修改。如果代码的异味很重，那么很常见的一种情况是：一个程序员尝试做一个应该很简单的修改，却不得不修改十好几个地方或者在代码文件中跳来跳去，并最终因为大脑无法跟上这不断扩展的环环相扣的修改而不得不放弃。

在一个开发人员进行*枪伤手术*时，他常常会发现代码中还存在另外一种代码异味：*半成品代码*（half-baked code，或者半成品对象，这在面向对象编程中很常见）。所谓的半成品代码，是指当程序员需要使用某一个对象（或者模块、单元等）时，必须同时对其他对象以特定的、事先规定好的方式进行初始化。比如，在初始化了某个库之后，你还必须为某些特定的变量设置默认值，同时还必须初始化它所依赖的另外一个库。你知道应该这么做是因为有文档或者示例代码；如果初始化不正确，就会导致程序崩溃，或者更糟糕，产生不可预测的行为。

注 3：我们强烈建议每个读者都多多了解不同种类的代码异味和反模式。在我们看来，这方面信息的最佳来源渠道之一依然是 Ward Cunningham 最初搭建的 Wiki（http://www.c2.com/cgi/wiki?CodeSmell）。

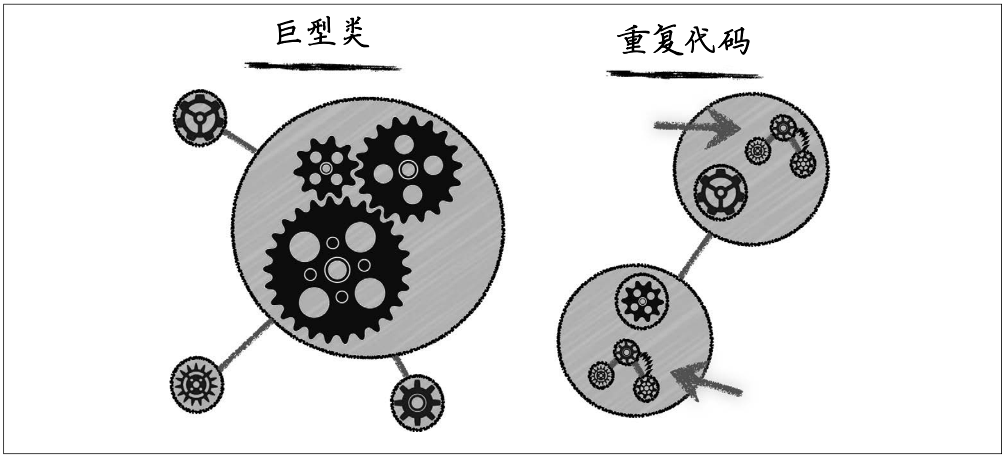


图 7-1：高效极限编程开发人员习惯于找出并修复像巨型类和重复代码这种代码异味。这些反模式往往与某个独立的单元相关。我们将在本章的图表里用齿轮来表示代码

某些代码异味与代码本身的写法有关。当你的代码中存在*巨型类*（very large class）或者非面向对象代码中的巨型方法、函数或模块等时，代码会很难阅读和维护。而更重要的是，这通常表示你的代码所做的事情过多，可以拆散成多个更小的、更容易理解的单元。另一方面，*重复代码*（duplicated code）是指一段雷同的（或者几乎雷同的）代码在多个地方出现。这很可能成为 bug 的源头，尤其是当某个程序员修改了其中一个地方，却忘了对剩下的地方做修改时。

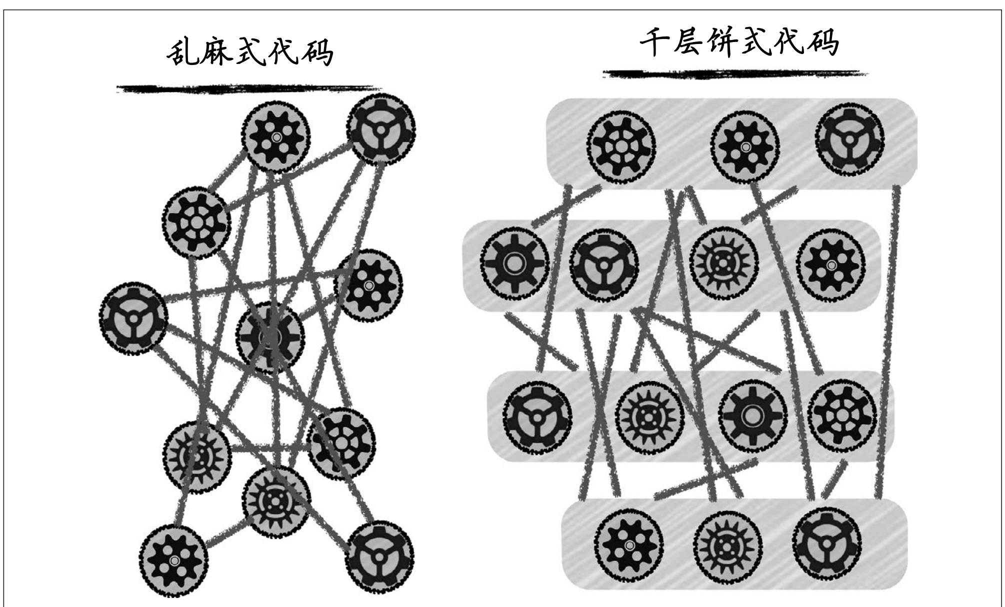


图 7-2：有些代码异味与系统更高层的设计以及各个单元之间的交互有关

其他的代码异味涉及代码的整体设计（即代码的各个单元之间是如何交互的）。*乱麻般的代码*，或者说有着复杂而混乱结构的代码，是软件工程界最为古老的代码异味之一，甚至可以追溯到 20 世纪 60 年代。乱麻式代码很容易地识别，因为它常常是曾经尝试过清理该代码但最终失败的开发人员写的恐吓或致歉注释。乱麻式代码的小兄弟，*千层饼式代码*（lasagna code），是更隐蔽的一种代码异味。现代软件设计一般会把代码进行分层，每一层有其特定的目的或行为。但是，当层级过多，而且各个层级缺乏一致的模式时，要理解每个层级要做什么就变得困难起来。这可能又伴随着层级之间的*代码泄露*，即，本应封装在一个层级的代码、类型或概念却泄露到临近的层级中去了。

7.2.3 钩子、边界情况以及功能过多的代码

我们刚刚描述的代码异味都是与代码的结构相关的反模式。还有一些问题是与代码的功能相关的。高效的极限编程开发人员明白，代码行为与代码结构同样值得关注。

举个简单的例子。有时候你可能预期某个代码单元（比如一个 Java 类）将来可能会以某种方式被用到，所以就给它加了一个*钩子*（hook），也就是一个用来处理将来某种情况的占位符。这个钩子看起来似乎是“无成本”的（因为你几乎没干什么），但是它其实是有代价的：你现在被一个决定限制住了，而这个决定本可以以后再做的。真到了需要用到这个钩子的时候，你对需要解决的问题已经有了更深入的认识，而这个钩子很可能需要修改。当你发现你当初对于钩子的使用做了一些预设，却最终发现后来你又在软件的另外一个地方绕过了那个钩子，那将会使你非常沮丧。这样一来，修改这个钩子变得更难了（尽管它根本就是空的），因为已经有其他代码在使用它了。一个习惯于添加很多钩子的开发人员会忘记那些钩子在什么位置。她会反复遇到这样的问题：她想要去用自己几周前编写的代码，却发现那些代码当初根本就没写，只是敲了个“TODO”而已。这是一个反模式：添加了太多的钩子，结果代码到底在做什么反而闹不清楚了。

另一个可能导致代码难以理解的反模式是对边界情况过于着迷。*边界情况*（edge case）是指那些很少出现或仅在特定条件下出现的情况。比如，一个从文件中加载数据的程序需要处理包含该文件的目录不存在的情况，通常会对此执行与正常情况下不同的行为。还有一种情况是，文件存在，但是无法读取，或者读到一半文件被删除了，或者文件的编码是错误的。仅仅针对这个简单地从文件中加载数据的场景，一个优秀的程序员就可以想出一大堆的边界情况。那么，你该如何取舍呢？

很多开发人员深陷边界情况的泥潭无法自拔。有些时候确实需要处理边界情况，很多情形下也确实需要做一些特殊处理。但是每新增一个边界情况，你得到的回报就越小。有些程序员经常会进入一个误区，即花费跟编写普通代码相当的时间去编写针对边界情况的代码，美其名曰：“增强代码可靠性。”这时，对边界情况的考虑就变得与过度计划相似，同时开始转移你的注意力。说到底，处理边界情况的代码很少运行。如果代码中充斥着各种边界检查，那将使得它更加难以理解，也就更加难以修改。

优秀的开发人员倾向于成为那种对细节着迷的人，而边界情况正是这种细节。聪明的头脑可以把一个优秀开发人员变成一个痴迷于边界情况的极端分子。一旦这种情况发生，团队会议将会被激情四溢且永不止息地对边界情况的讨论所劫持，屋子里每一个开发人员都突然强烈地认为他或她提出来的不太可能发生的边界情况必须要得到处理。

当开发人员对边界情况过度计划或添加过多的钩子时，他们就是聪明过头了。这个时候他们想的不是如何编写简单、易修改的代码，而是编写那种过度复杂、过度抽象或者解决明天而不是眼下问题的代码。

这就要说到我们称之为框架陷阱（framework trap）的代码异味了，这是一种源自于程序员聪明过头的反模式。所谓框架陷阱，是指当一个程序员需要解决一个问题或执行一项任务时，他不是写代码去完成要做的事情，而是编写了一个更为庞大的、可以在将来被用来解决类似问题或执行类似任务的框架。

Ron Jeffries 是与 Kent Beck 共同发明了极限编程的人，他曾经描述过一个简单的避免框架陷阱的办法："只有在你真正需要一个东西的时候才实现它，永远不要在你仅仅是预见可能会需要它的时候实现它。"[4] 有些极限编程团队在谈及这个问题时喜欢用 YAGNI 这个缩写，即 You Ain't Gonna Need It（你不需要这个）。当你尝试预见一个需求，然后编写代码来满足那个需求时，聪明过头的你就正在掉进框架陷阱中。

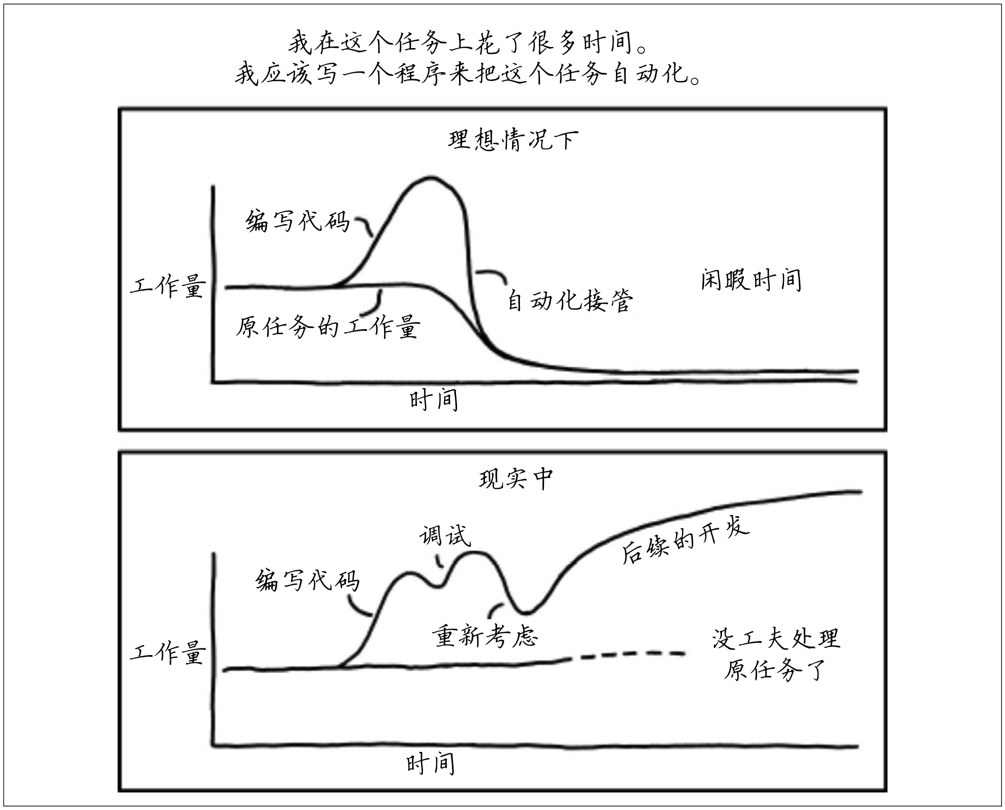


图 7-3：知名的在线漫画网站 xkcd（http://xkcd.com/1319）对开发人员掉进框架陷阱的情形所做的精彩展示

注 4：见 YAGNI 的维基百科页面（http://en.wikipedia.org/wiki/You_aren't_gonna_need_it），该页面加入维基百科的时间为 2014 年 7 月 6 日。

相比解决眼下单一、具体的问题，聪明的开发人员更喜欢把目标定得更高一点。他们可能会这样想，“我能写出这个简单的解决方案，但是如果我能把这类工作自动化了，其他人以后就再也不用花心思解决相同的问题，这不是更好吗？”结果是：原本是要开发一个网页，结果常常是做出来一个生成网页的框架；原本是要解决一个特定的性能问题，结果却搞出来一个庞大、通用的缓存模块；日程表上一个下载文件的简单程序，最后不知怎么居然包含一个脚本引擎。

要理解开发人员为什么会走向这样一条道路并不难。他们可能是这么想的：“要是有人针对我现在正在做的这个东西编写过一个框架，那我直接拿来用就行了。既然如此，那我为什么不多花点时间，思考一下如何对问题进行抽象，然后找出一个更加通用的解决方案呢？”这其实是一种值得尊敬的想法，同时它也解释了为什么这么多团队都走入了框架陷阱的误区。

1. 可是，可复用的框架有什么问题吗？

可复用的框架没有任何问题，极限编程也绝不排斥对框架的开发和使用。事实上，我们最受欢迎的书之一《深入浅出 C#》恰恰是教程序员如何使用微软的 .NET 框架的。每个平台上都有一些非常优秀的框架，用来构建 Web 应用，进行 3D 图形渲染，开发网络服务，等等。

但是这并不意味着你的项目就应该开发一个框架。

咱们就先谈谈可复用性这个问题，毕竟框架陷阱指的是编写可复用代码时的一种反模式。*可复用代码*也就是可以在系统的多个地方反复使用的、有一定的通用性的代码，它们是开发人员的奋斗目标之一。这当然非常有意义。文档处理器中对话框里的按钮，无论行为还是样式都很相似。如果对每个按钮都单独写一套代码，那效率实在是太低了；这时开发人员会为这些按钮写一份代码，然后在每个对话框中对该代码进行复用。可是，文档处理器的对话框按钮跟网页浏览器中的对话框按钮也基本是一样啊，不仅如此，几乎所有应用程序中的对话框按钮都基本一样。所以，实现这些按钮的代码（很大的可能）实际存在于所有这些程序的外部，这些基本按钮是由操作系统的库提供的。按钮相关的代码只写了一次，但可以被无限复用。这种做法显然节省了大量的时间。

开发人员很少从零开始开发什么东西。他们有各种各样的库可以使用：读写文件的、访问网络资源的、绘图的，凡此种种，几乎每一项功能你都能找到相应的库。这是编程的一项常识。事实上，当一个优秀的程序员遇到一个他没有解决过的问题时，他常常会先上网搜一下，看是否有某个库可以解决该问题。因此，当一个程序员解决了一个别人也可能遇到的问题时，他的第一反应常常是想办法把他的代码分享出去[5]。这是软件开发中一种正常的、有助于提高生产效率的思路。

2. 库和框架的区别

如果你能把一个东西作为单一的独立部分整合到项目中，而不必为此引入很多其他组件，那么这个东西就是一个库。库和框架的区别在于，库是要把代码拆散成一个个小的、可复用的组件。如果你的目标是写出简单的代码，那么每个组件就应该只做一件事；功能繁多

注 5：我们自己就曾经做过这样的事。到我们的网站（http://www.stellman-greene.com）上看看，你会发现我们发布的一些开源库，这样其他程序员就不用去解决我们已经解决过的问题了。

的组件应该被拆分成多个更小的单元。这种拆分称为关注点分离（separation of concerns），极限编程开发人员都很清楚，这是实现代码简化很重要的一点。

另一方面，框架的作用在于把很多小的、可复用的部分组合成一个大系统。比如，学习 C# 往往意味着你不仅要学习 C# 的语法，还要学习如何使用 .NET 框架。它包含了诸如文件访问、数学计算、网络连接以及很多其他功能的库（没错，也包括按钮）。当你使用 C# 和 .NET 开发程序的时候，这些东西对你来讲都是“免费的”，因为它们是 .NET 框架的一部分。一般来说你是无法把它们排除出去的。况且，在一个非 .NET 程序中使用 .NET 的一个部分似乎也没什么意义。在这里，像大部分框架一样，众多的组件被组合为一个要么照单全收、要么啥也没有的大系统。

框架要比库复杂得多，所以在更应该开发一个库的时候却开发了一个框架常常会导致过度复杂的设计。有些程序员看到其他的框架节省了他们很多时间，于是希望他们自己的代码也能做到这一点，这个时候上述错误就比较常见。这些程序员会说：“我没有引入任何的库，就可以在我的 .NET 程序里使用这个按钮，这一切都是‘免费的’。如果我把我的整个系统也做成一个框架，那我也能给其他开发人员提供好多‘免费的’东西！”

这么处理问题的一个缺点在于，如果你想用那些“免费的”东西，你必须使用整个框架。而这会让你从不同的角度思考问题。一个 .NET 开发人员看问题的角度往往跟一个 Java 开发人员不同（当然也不至于大相径庭，因为 Java 和 .NET 的框架有很多概念是相通的），也跟 C++、Python、Perl 或者 Lisp 开发人员十分不同。这是因为，在面对一个问题的时候，如果你手上有一组工具，你会从这些工具着眼去寻找解决方案。这些例子都是语言和它们的框架（用 Java、Python 或者 Perl 开发软件时，你一般不会不去想它们的标准库），因此它们不会对你思考问题的方式造成太多的限制。

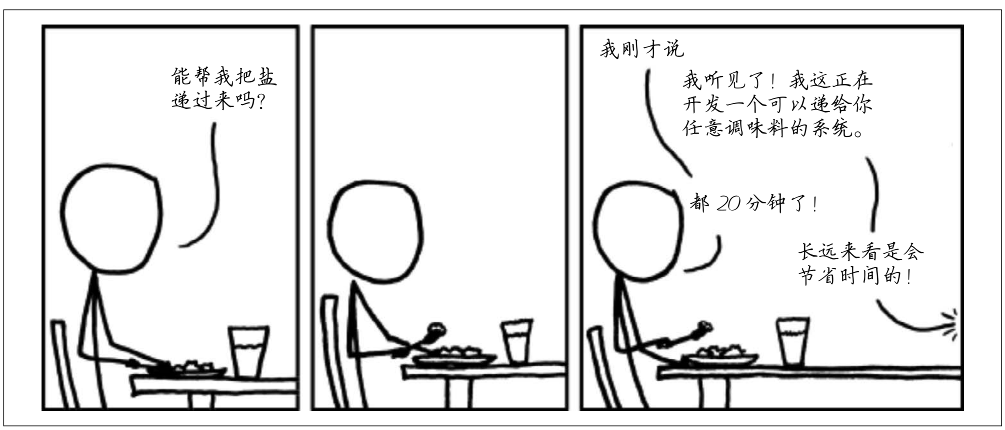


图 7-4：有时候，为解决简单问题而开发复杂的框架似乎是正当的

举一个常见的框架思维的例子。开发团队会使用脚本来自动对软件进行构建。（在第 6 章中，我们已经学习过，极限编程团队的构建脚本运行时间不超过 10 分钟。）编写这些脚本的工具很多，多数都包含脚本语言。但是对于有着框架情结的开发人员来说，为每个项目编写各自的构建脚本是不够的。因为这些脚本中重复的部分太多了，因此，他会为所有的

项目写一个构建框架。（“使用了我的框架，你再也不用自己写构建脚本了。我的框架直接替你做了。”）于是，原来的每个项目一个小脚本变成了一个需要单独维护的庞大的框架。这个程序员在这里做了一个取舍：他避免了多个项目中构建脚本那几十行代码，但是换成的却是一个有着几百上千行代码的框架。一旦构建出现问题（根据墨菲定律，这通常发生在软件发布的前夜），那么需要调试的就不是一个小小的构建脚本了，运行构建的开发人员不得不在一个大型的框架里尝试定位问题。这与简化的思想是相悖的。

把大量功能组合成一个大型的框架也许能够非常有效地解决某个特定问题，但是当问题本身发生变化时，框架就会使其更难应对变化。

换句话说，通过组合众多的组件构造出框架，本意是希望能够节省时间，但是造成的结果却是给自己套上了枷锁，对于新信息和新需求的响应受到了框架的限制。我们把这叫作“框架陷阱”，这对于理解“简化”这一思想及其对项目的影响是很重要的一个概念。我们之所以拿它来举例，是因为它反映了与极限编程相抵触的思维的一个重要方面：偏爱把功能组合成单一的大单元，而不是拆散成众多的小单元。

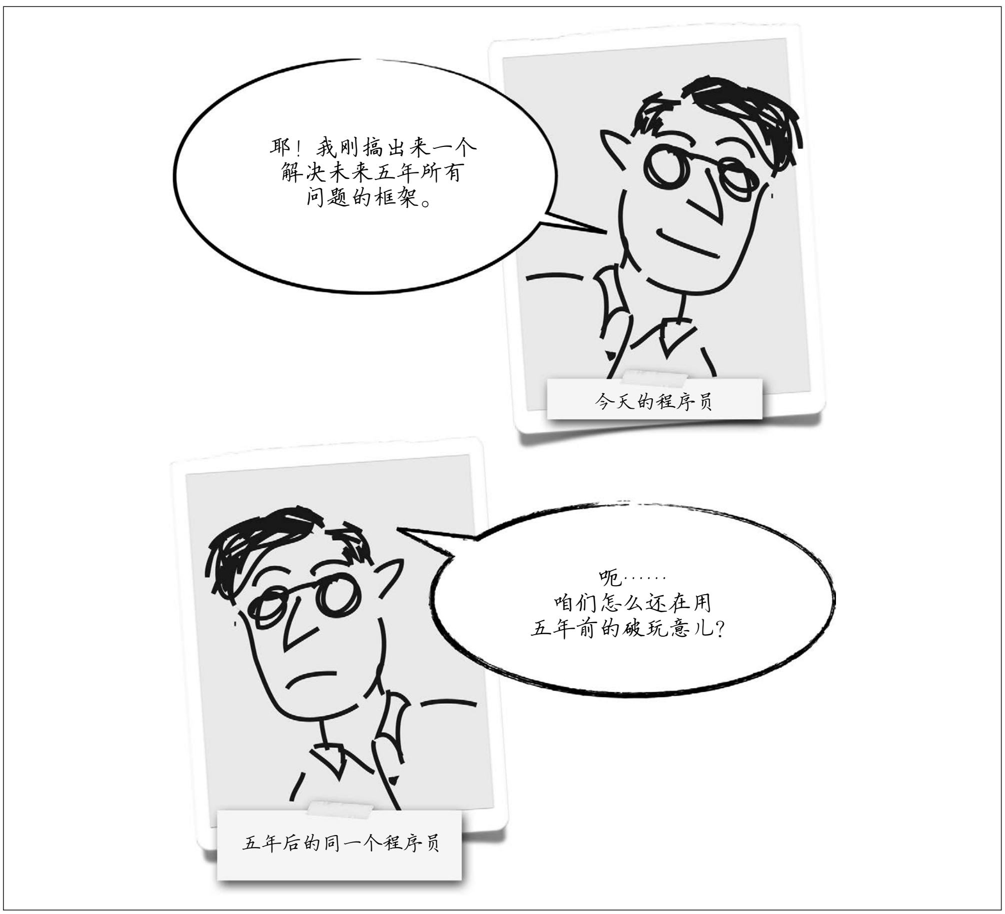


图 7-5：今天看似美好的框架，明天就可能成为负担

7.2.4 代码异味会增加复杂性

我们刚刚描述了一些极限编程团队会习惯性地避免的代码异味。这些代码异味的一个共同点是，它们会增加项目的复杂性。反过来也是一样：如果你发现你必须记住一个对象应该如何初始化，或者必须费劲地去记住一层又一层的修改链条，或者无法忍受在让人痛苦的乱麻般的代码里寻找问题，那么这些都说明你的代码几乎肯定存在着大量不必要的复杂性。因此，极限编程团队会努力尽早地找到并修复这些代码异味。

我们提到的这些代码异味并不是全部，甚者它们可能都不是你的项目中最常见的那些。但是，经过多年与众多开发人员的沟通，我们发现几乎每个人都在自己的代码中见过这些问题中的至少一个。这些问题的产生并不是因为开发人员技术水平不行，也不是因为出于懒惰或者恶意的主观故意，更不是平白无故的。它们是开发人员养成的坏习惯所造成的结果，尤其是当开发人员时间紧迫时，或者他们被强迫克服某些限制的时候（尤其是那些他们之前自己引入代码中的限制）。

那么，当你发现自己的项目中存在代码异味时，该怎么办呢？能否有效地预防它们呢？

要点回顾

- 极限编程团队会及时发现并清理**代码异味**，也就是代码中的**反模式**，以防止过度复杂并保持设计的简洁。
- **半成品代码**和**巨型类**这样的代码异味可以帮助团队找出那些可以进行简化的独立单元。
- **千层饼代码**这种异味可以帮助团队找出更高层次的设计问题。
- 尤其让人头疼的一种代码异味称为**枪伤手术**，指的是当一个开发人员尝试修复一个问题时，发现要想修复它还必须修复代码库的其他部分，而这些部分又需要更多额外的修复。
- 那些针对边界情况做过度的计划，添加过多的钩子，或者开发大型框架来解决个别问题的团队有些聪明过头了，结果导致的是过于复杂的代码。
- 即便是技术过硬的团队，如果没养成好习惯，也可能会犯这些错误。

7.3 把编码和设计决定留到最后责任时刻

> 从传统设计向极限编程式设计的转变，就是做决定的时间的转变。设计决定被推迟到有了足够的经验且该决定能够马上被用到的时候。这就使得开发团队能够做到以下这些。
>
> - 更早地部署软件。
> - 做决定的时候更有把握。
> - 避免被错误的决定束缚手脚。
> - 在原有设计前提发生变化时保持开发速度。
>
> 这种策略的代价在于，开发团队必须在整个项目的生命周期里持续地投入时间和精力去做设计，并以小步慢走的方式去做大的变动，这样才能保证不断地交付有价值的新功能。
>
> ——Kent Beck，《解析极限编程：拥抱变化（第 2 版）》

我们已经知道，Scrum 团队在最后责任时刻才做项目计划的决定。这使得团队的计划更加简单，对团队来说，整个计划就是一些用户故事，还有贴在任务板上的任务卡，再加上一个积压工作表。团队的项目会议也简单得多，会议可以加上时间限制，因为只有那些相关的信息才需要讨论。

换句话说，Scrum 团队利用最后责任时刻的概念简化了项目计划的过程，同时也有助于提升工作效率，并且通过避免项目管理中的反模式让团队能够作出更好的决定。

对于项目中技术方面的计划，极限编程团队的做法也很类似：把“简化”这一核心价值应用到架构、设计以及编码上。像 Scrum 团队一样，极限编程团队也把架构、设计以及编码的相关决定留到最后责任时刻。大部分情况下，这个时刻是在代码已经被写出来之后。

设计决策居然在代码已经被写出来之后才做，这听起来是不是有点奇怪？这对极限编程开发人员来说并不奇怪，因为极限编程团队不断地重构其代码：即，修改代码的结构但不修改其行为。重构并不是极限编程的专利；它只不过是一项为极限编程团队所采用的、普遍且有效的编程技巧。事实上，大部分的 IDE（用于编辑、运行和调试代码的软件）都有内建的重构工具。

我们举一个重构的例子，即使你不是开发人员，你也能够看出重构是如何把代码变得更加清晰易懂的。在写作关于 C# 编程的《深入浅出 C#》一书时，我们曾用下面这段代码作为其中一个读者练习项目的解决方案（该项目是一个蜂箱模拟器，所以变量名都跟蜜蜂有关）。

```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl)) {
            fieldForm.Controls.Remove(beeControl);
            beeControl.Size = new Size(40, 40);
            hiveForm.Controls.Add(beeControl);
            beeControl.BringToFront();
    } else if (hiveForm.Controls.Contains(beeControl)) {
        hiveForm.Controls.Remove(beeControl);
        beeControl.Size = new Size(20, 20);
        fieldForm.Controls.Add(beeControl);
        beeControl.BringToFront();
    }
    beeControl.Location = bee.Location;
}
```

这四行把一个BeeControll从Field表单移动到Hive表单

而这四行把一个BeeControll从Hive表单移动到Field表单

图 7-6：这是我们在《深入浅出 C#》一书某个项目中的原始代码片段

在技术审校期间，有一个审校人员指出这段代码太过复杂了，他感觉这个方法有点过大了。于是，我们做了一个大部分极限编程团队都可能做的事情：我们对这段代码进行了重构，使之更加简单，易于理解。具体地，我们把四行代码提取出来，放入一个名为 `MoveBeeFromFieldToHive()` 的方法中。然后对另外四行代码做了相同的修改，把它们提取到一个名为 `MoveBeeFromHiveToField()` 的方法中。该书交付出版社的时候，这段代码变成了下面这样（那两个新方法的代码在后面）。

```
foreach (Bee bee in world.Bees) {
    beeControl = GetBeeControl(bee);
    if (bee.InsideHive) {
        if (fieldForm.Controls.Contains(beeControl))
            MoveBeeFromFieldToHive(beeControl);
    } else if (hiveForm.Controls.Contains(beeControl))
        MoveBeeFromHiveToField(beeControl, bee);
    beeControl.Location = bee.Location;
}
```

图 7-7：我们通过提取两个方法的形式重构了代码。新代码更加简单，也更容易看出它在做什么

这样就清楚多了。重构之前，要理解该代码在做什么，需要了解更多关于整个程序结构的信息，所以我们不得不添加一些手写的标注来帮助读者理解那两个代码块。把这两个代码块提取到各自的方法中，并给每个方法起个合适的名字，就使得代码的功能更清楚了。在重构后的版本中，你能够一眼看出那两段代码在做什么：一个是把蜜蜂从野外移动到蜂箱里，另外一个则把蜜蜂从蜂箱放回到野外。

这次重构使得该段代码更加容易理解了。与此同时，它还降低了整个项目的复杂性。很有可能在项目的其他地方，某个程序员也需要把蜜蜂在蜂箱和野外之间移动。有了这些方法，那么他就更有可能直接使用它们，因为这么做对他来说是最简单的。而且，即便他一开始没有直接使用这两个方法，如果他后来发现出现了重复代码，他也更可能做一个快速的重构，去掉重复的代码，用两个方法调用予以替换。

7.3.1 决然重构，偿还技术债务

> *发布新代码就想借贷一样。背一点债务可以加速开发进程，只要能够及时地通过重写把债务还清就行。可是一旦债务没有被还清，危险就会来临。*
>
> ——Ward Cunningham，*敏捷宣言起草者*

差劲的软件设计会随着时间推移而聚沙成塔。就算是杰出的开发人员所写的代码也有改进的空间。设计和编码上的问题在代码中存在的时间越长，这些问题就越会累积，最终导致枪伤手术那样的麻烦。一般开发团队把这些遗留的设计和编码问题称为*技术债务*（technical debt）。高效的极限编程团队会在每个周期中留出专门的时间来还债。这是*丢车保帅*（即在每个周循环中加入一些次要的故事和任务，从而为计划外的工作留出缓冲空间）的一种很好的用法。

任何一个好的财务顾问都会告诉你，最好的避免财务问题的方法就是不要欠债。这对于技术债务一样适用。这就是为什么极限编程团队会*毫不犹豫地重构*，从不间断地在其代码中寻找代码异味，并想办法通过重构来简化代码。随着重构的不断进行，团队会更多地了解到其代码是如何被使用的，以及这种真实的用法与其预期有哪些不同。通过不断地修订，代码库中的每个单元都不断得到改进，变得更加适应真实的使用场景。这种不断编码并进行修订的做法本质上就是迭代性质的，所以很多团队并不会就此在项目开始时进行有意识的计划。尽管毫不犹豫的重构会花费额外的时间，但它节省的时间远比它消耗的多，因为

一个简单的代码库比一个复杂的代码库要容易维护得多。

当一个团队中的每个人都养成了不断重构的习惯时，他们写出的代码就变得更加容易修改。当团队成员发现他们需要实现一个新故事，或者更常见的，发现他们之前对某个故事的理解有错误，需要修改现有代码，这种修改会更加容易。他们将能够拥抱变化（这是每个极限编程团队的一个首要目标），因为他们不需要为了做出修改而与现有代码进行对抗。

1. 重构不是相当于重写吗？重写不是bug的主要来源吗？

没错，重构就是重写。也确实，重写容易引入 bug。如果项目后期用户的需求发生变化，开发团队在进行相应的修改时可能会不小心引入 bug。但是重构虽然是重写，却可以防止 bug。不断对项目进行重构可以让你的代码库由更小的、天然可复用的单元组成。而与此相对的是绝大多数有经验的程序员所熟悉的那种难以修改的代码库。

重写在传统的瀑布式项目中是一个主要的 bug 来源。一个原因是，随着设计变得复杂，代码变得问题更多、更脆弱，也更加难以修改。但是，如果重构造成开发人员由于疏忽而引入了 bug 呢？比如，一个代码块做两件不同的事情，而开发人员却把它提取到一个只做其中某一件事的方法中，怎么办？极限编程团队给出的答案是：测试先行（或者测试驱动）开发，这是极限编程的主要实践之一。当一个程序员已经有了针对待重构的那个单元的一组测试，重构起来就安全多了。事实上，开发人员会对大幅度的重构感觉更放心，而这种大幅度重构在没有测试的情况下会感觉太过激进、风险太大。因为重构的定义就是改变代码结构而不改变代码行为，相同的测试应该在重构前后都能通过，而在实际开发中，测试确实能够捕获几乎所有原本可能通过大幅重构而被引入的 bug。

2. 难道不应该从一开始就用优秀的设计避免这类问题吗？一开始就把代码写正确不是更安全的选择吗？

是的，最安全的就是一次性把代码写对。这也确实是软件工程（尤其是在需求分析和项目管理领域）曾经追求多年的主要目标之一。但是，第一次就把代码写对的可能性是非常小的，因为团队对于问题的理解会随着项目的推进和代码的增加而发生变化。这种变化是很正常的，是团队持续交付可工作软件的自然结果（而我们知道与详尽的文档相比，可工作的软件是更好的评估工具）。

也正是因为这种变化是自然存在的，极限编程团队使用迭代式的开发过程，并且在主要实践中包含了季度循环和周循环。团队在每周的迭代中都给自己足够的时间来编写单元测试并进行代码重构。而每一次可工作软件的交付都帮助团队与用户一起增进他们对正在解决的问题的理解，并改进故事。通过不断地寻找代码异味并改善代码设计，团队就可以写出容易修改的代码。

我们也不必对从前那些通过大需求和大设计先行方式开发软件的团队太过苛刻。那些团队当时并没有成熟的工具（这里的“工具”既指软件工具，也指后续发展出来的团队实践）进行方便的重构和单元测试。那时，甚至开发和发布都很不容易，但是编译代码就要耗费好几天甚至好几个星期，而且当时的电脑也没有联网，所以软件都是复制到 CD、软盘、甚至磁带上。这些都给重写设置了非常昂贵的基本开销，因此可以很容易地得出结论认为开始编码前值得投入资源去编写文档并进行严格的审查。

7.3.2 持续集成，排查设计问题

我们在第 6 章学习了“持续集成”这项实践，它是开发团队们现在可以使用的成熟工具之一。持续集成能够改进设计并防止设计问题出现的原因之一是，它可以让团队在引入集成问题的时候就遭遇失败，而不至于等到过了很久才发现问题。

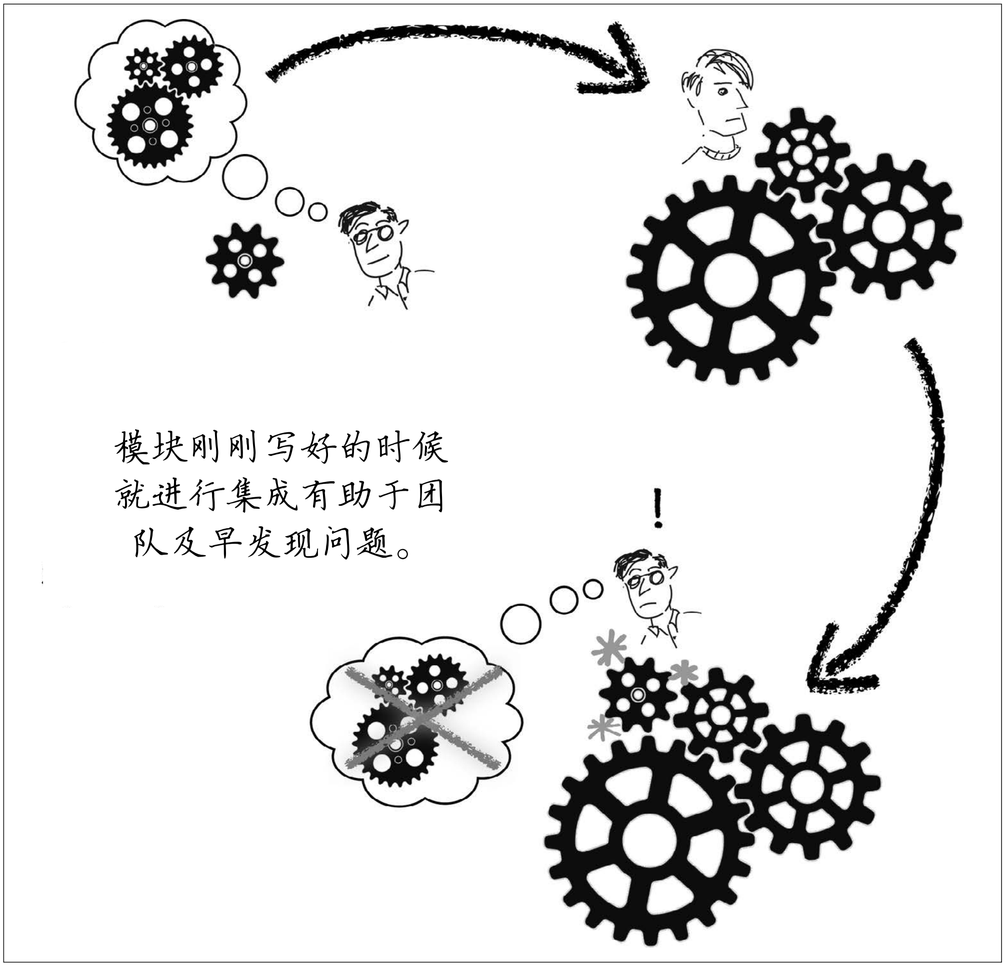


图 7-8：持续集成可以及早发现问题

一个被设计用来在发生问题时第一时间马上报警的系统被称为*快速失灵系统*。如果你的系统需要容错，而且发生错误时需要尽快的找出其根源，那么你会希望你的系统快速的失灵。早一点失灵可以提供一个反馈循环，让你把获得的知识再回头应用到项目中去。这个思想既可以应用在软件产品上面，也可以应用在团队开发软件的方法上。

*失灵*是我们在第 6 章中讨论的极限编程的指导原则之一，而使用持续集成可以让项目能够在两个团队成员添加了不兼容或相冲突的代码时快速失灵。认同极限编程的团队成员会把

集成失灵的情况看成是正面的，因为这种失灵可以帮助团队及早的找出并修复问题，这个时候问题还比较容易修复。



图 7-9：设计上的问题如果及早发现，是比较容易修复的，也可以避免未来的麻烦。这是持续集成带来更好总体设计的一种方法

测试先行编程帮助团队自然而然地写出较小的、独立的、容易集成的模块，这样持续集成对于极限编程团队就不再是一项负担。团队不断地测试每一个单元以确保每一个部分都能正常工作。当团队中的每个人都持续地把它们的代码与整个代码库进行集成时，就可以防止单个开发人员自行其是，开发出一个与系统格格不入的模块。

7.3.3　避免一体式设计

假如说你正在开发一个软件，而开发到一半，你对该软件需要解决的那个问题的认识更进了一步。这对大多数软件团队都是很熟悉的场景，而你的团队需要具备在最后责任时刻做决定的能力才能解决这一问题。

传统的瀑布式团队经常对此一筹莫展。在项目开始前把需求确定下来，经过很多人的审查，然后一次性把代码写好，这种开发方法天然地会导致代码难以修改。开发团队本身也没有什么动力把系统设计得易于修改，因为改动是被严格控制的。开发人员也从未养成像随时重构或发现代码异味这样能够帮助他们写出容易更改的代码的习惯。

这常常导致一体式的设计（monolithic design），即：由庞大的、互相交织在一起的、有为数众多的相互依赖的、难以分离开来的单元所组成的一种软件设计。

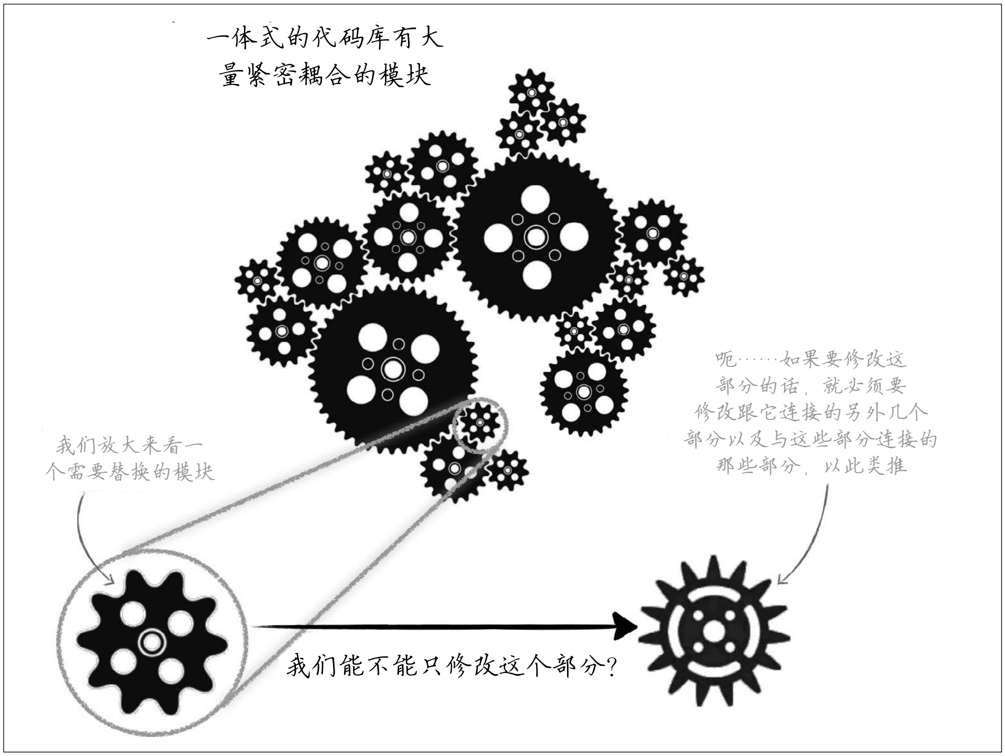


图 7-10：如果开发人员没有养成那些帮助他们写出简单、松耦合代码的习惯，他们写出的代码就会是一体式的

本书前几章中，我们谈到了软件开发与建筑之间的不同。对于建筑来说，最具破坏性的事情莫过于拿一把大锤子把墙砸倒。在软件开发中，删除代码造不成什么破坏，你可以随时从版本控制系统中把删掉的代码恢复出来。在这里，最具破坏性的事情是写出糟糕的代码，然后再在它的基础上写出更多糟糕的代码。添加一个这样的依赖有时候称为“耦合”，因为一旦这两个部分的代码被联系在一起，那么要想修改一个而不影响另一个就没那么容易了。

一体化设计常常伴随着*紧耦合的代码*（tightly coupled code），也就是每个单元都跟很多个其他单元有联系。修改高度耦合的代码是很让人头疼的事情，耦合常常导致枪伤手术以及很多其他反模式。而各个单元之间的联系（常常是没有文档说明的）将导致重写过程中引入 bug。

正如代码可以耦合，它也可以进行*解耦*（decouple），也就是打破单元之间的联系（或者更好的一种做法是，一开始就不加入这种联系）。

发现代码异味并进行重构可以帮助你写出解耦的代码。让我们回到重构蜂巢程序的例子，我们在该例中把一部分在蜂巢和野外之间移动蜜蜂的代码提取成了两个方法。我们看到，这两个方法可以在代码库的其他需要相同功能的地方使用。可是，如果那两个方法是某个需要做大量初始化的大型单元的一部分呢？如果一个开发人员有着良好的习惯，那么当他

尝试复用这两个简单的方法时，就会发现这个半成品代码问题并对它进行重构，去掉多余的初始化代码。这样，它就可以被代码库中两个不同的部分所使用，而且不需要知道它是如何被调用的，因为它已经与调用它的代码解耦了。如果还有第三个地方需要调用它，它跟那个部分之间也已经解耦了。

当一个系统由小型的、独立的单元构成时，是最容易维护的：每个部分的代码都与其他的部分尽可能地解耦（有时候能够做到的解耦程度令人吃惊），从而把彼此间的依赖关系降到最低。这是构建一个不需太多的返工就能够修改的系统的一个关键。如果你的设计是高度解耦的，那么你很少会去做枪伤手术或在一次变更中同时修改系统的多个部分。如果你已经养成了随时重构（而不是等到以后）以减少耦合的习惯，那么你的代码库会处于一个比以前更好、更加松耦合的状态：每个代码单元只做一件事，并且与不相关的其他单元解耦。

7.4 增量式设计与极限编程的整体实践

在第 6 章中，我们讲解了 13 种主要的极限编程实践中的 10 种，并把它们分别归入了四个类别：编程实践、集成实践、计划实践和团队实践。还有 3 项主要实践，我们把这三项归入另外一个类别：整体实践。我们之所以这么晚才讲到它们，是因为只有把目前我们所覆盖到的内容全部考虑进来，它们才真正有意义。既然把它们称为“整体实践”，说明它们是紧密相关的，没法独立起作用，你没法只做一项却不做其他两项（与此相对的，像结对编程或者周循环，这些都可以独立起作用）。

我们要讨论的第一个极限编程的整体实践是*增量式设计*（incremental design）。在所有的极限编程主要实践里，这一项对刚接触极限编程的人来说是最难理解的。本章的一个目标就是帮助你理解这项实践，以及它是如何影响整个项目和团队的。

你能够在很多成熟、高质量的开源项目（如 Linux、Apache HTTP 服务器和 Firefox）中发现增量式设计的好例子。这些项目都是围绕着一个坚实的核心进行设计的；开发人员使用一种插件式的架构（或者其他把代码与核心隔离开来的方法）来开发附件功能，而只有那些最常用的、最稳定的功能才会被纳入到核心中去。这样的设计带来的就是高度解耦的代码，这些项目的开发人员也习惯于不断重构、编写大量的测试，并且持续地进行集成。（事实上，很多极限编程的实践都或者源自于开源项目，或者经过开源项目的打磨。）

这些开源项目的开发方式都是这样的：开发人员编写相对独立的松耦合的单元，这些单元能够与核心协同工作，而整个项目的代码库则围绕着核心有机地生长着。每个开发人员一般都是独立地添加他或她自己的单元，并且假定所有人编写的代码都是高度解耦的，这样就可以保持新单元与已有单元之间发生冲突的几率较低。但是，这不是凭空做到的，团队成员都很清楚这一点。所有对现有代码的丰富都发生在同一个代码库上，所以这些团队都有自动和手动两种持续集成机制。（今天的某些持续集成和构建服务器实际上正是源自于这些项目。）同时，每个开发人员也都对代码异味和反模式高度警惕，并且感觉到有责任在发现它们的时候尽快予以修复。重构几乎从来不会被留到项目的最后；每个程序员都感觉到一种责任，只把他或她写的最好代码添加进来，并依靠其他开发人员的眼睛来不断发现和纠正问题。

正是由于这些优秀的实践和开发人员习惯，项目的代码库可以随着时间的推移增量式成长；这就是一个大型的、分布在多个国家的团队编写优秀软件的方式。[6]

从根本上讲，增量式设计的核心在于“把设计决定留到最后责任时刻”，避免落入“一次性全部完成”的陷阱，该陷阱是很多开发人员（甚至包括很资深的开发人员）经常犯的错误。

落入上述陷阱的团队养成了一些坏习惯，会导致一体化的或令人费解的设计。比如，团队可能更关注解决那些与其眼下问题不太相关的更大问题（比如关注更多的边界情况而不是代码需要实现的具体功能），或者过度考虑未来的情形并添加过多的钩子，或者构建大型的抽象框架来解决小而具体的问题。这些习惯有什么共性呢？它们都在项目的早期作出了过多的设计决定。

另一方面，当团队建立起编写由小型的、可靠的、独立的单元组成的松耦合代码的习惯时，就没有必要在最开始的时候就给出完整的设计。团队可以先给出一个高层次的设计，不去考虑每一种可能不适用这种设计的情况。团队有这样一种自信，即当它发现一种没有考虑到的边界情况时，它拥有解决和处理这种情况的工具。这给了人们以一种更加自然的方式思考问题的自由，并且允许他们随着对问题认识的深入，逐步地编写代码。这就是增量式开发背后的思想，这种思想会使代码库高度灵活、适应性强，而且容易修改。但是，它要求每一位开发人员真正地做到把设计和编码决定留到最后责任时刻。团队中的每个人都应该相信更快速的开发方式是编写小型的、松耦合的模块，同时大家也应相信如果这些模块写得有问题，他们将来能够修复它们。

这不仅仅关乎开发人员如何设计和编写代码。团队的风气也很重要：其互动方式、工作环境，还有更重要的，大家所持有的态度。

很多开发团队没有足够的时间来完成其工作。或者更糟，它是被人为因素搞得没有足够的时间的。截止时间完全是随意决定的，很可能是某个老板或项目经理拍脑袋想出来的，还觉得这种不可能的截止时间可以很好地激发士气。在这种团队中，任何不增加代码的工作都被看成是多余的。

如果你感到时间不足，那么无论你有多么好的习惯，恐怕都不重要了。来自工期和日程的压力越大，你就越有可能抛弃像重构、测试先行编程和持续集成这样的好习惯。比如，你可能不再重构了，因为代码能工作，而且你感觉没有时间去改进它。无论有没有单元测试，代码都能运行，所以，你会感觉在单元测试上多花一分钟，在功能代码上就得少花一分钟。我们心里知道这些实践能够提高开发速度；但是当面临巨大的压力时，潜意识会告诉我们“我没时间！”

注 6：关于开源团队如何工作，你可以从 Eric S. Raymond 的《大教堂与市集》一书中了解到更多内容。你可以从 Karl Fogel 的 *Producing Open Source Software* 一书中了解到很多开源团队的日常工作情况，该书可以在以下网址免费下载：http://producingoss.com。当然，我们建议从 O’Reilly 购买该书以支持他的杰出工作。

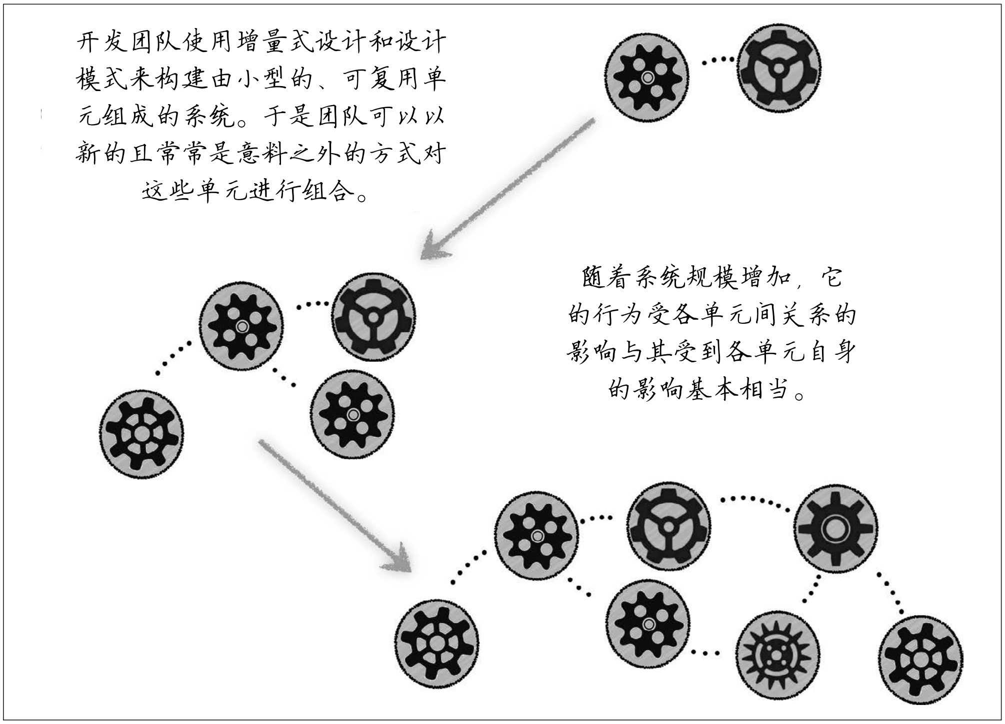


图 7-11：增量式设计能够让系统更可靠、可维护性更强

让我们来考虑下面这种情况：一名开发人员，他所在的团队面临着来自老板的巨大压力。可能是老板向用户承诺了一个不切实际的产品交付期限，于是老板为了赶这个工期，给团队施加压力，让团队不要编写单元测试，不要做重构，或者就干脆不要做任何不直接实现产品功能的事情。在这样一个团队中，开发人员常常会在遇到代码异味的时候这样想："我没工夫考虑这个！我应该尽快把这个功能做完，好开始做下一个。"他们会自然而然地写出紧耦合的代码（因为花时间解耦被看作是多余的工作，而且"我没时间！"）。他们的老板对于有效地采用极限编程是一个障碍，为这种老板工作的开发人员更容易写出问题多多、难以维护、难以修改的代码。他们的项目几乎一定会延期，有一些甚至可能直接以失败告终。[7]

极限编程对此也是有办法的。这就为我们引出了极限编程的最后两个整体实践：精力充沛的工作（energized work）和有凝聚力的团队（whole team）。

7.4.1 有时间进行思考，团队才能做好工作

软件开发是洞见的游戏，而洞见只会光顾那些有准备的、精力充沛的、放松的头脑。

——Kent Beck，《解析极限编程：拥抱变化（第 2 版）》

注 7：回头看看第 5 章 Grady Booch 有关创新和害怕失败的那段话。那是一个很好的反面例子。

*精力充沛的工作*这一实践的含义是，要创造一种环境，让每个团队成员有充分的时间和自由去做他的工作。这可以让他们保持一种良好的精神状态，去养成并发扬那些有助于写出更好、更容易修改的代码的习惯。反过来，这又使得他们能够在更短的时间内，写出更多的代码，为用户和客户交付更多有价值的功能（同时也是价值更高的特性）。

软件开发几乎完全是一项脑力劳动[8]。每一位优秀的开发人员都有过类似的经历：盯着一个问题研究了好几个小时也没有研究出个头绪，但是却在吃晚饭（或者冲凉、骑车等）的时候灵光一闪，想出了解决方案。每个软件项目都依赖一系列像这样的小创新。

有一点是广为人知的，那就是开发人员进入"状态"是需要一定的时间的（通常为 15~45 分钟）。进入这种状态时，她能够保持精神高度集中，并且保持较高的生产力[9]。任何打搅和让人分心的东西都可能打破这种状态。在交付代码的压力之下，开发人员是很难进入这种状态的，因为她一心急着要把代码赶紧写出来交差，没有时间去思考。

那些不现实的、拍脑门定下来的、对程序员缺乏尊重的工期，以及老板的一些其他不良做派会导致一种*无精打采*（unenergized）的环境，处在这种环境中的人们感觉自己无法做决定，而且没有精力去做好的决定或者去创新。

一旦形成了开发进度一直滞后、团队受制于不切实际的工期这种风气，人们就会在写代码的时候偷工减料。这通常意味着放弃极限编程带来的那些优秀实践。更重要的是，这意味着开发人员很少能够真正进入状态，而进入状态正是让项目得以简化的必要条件。

极限编程努力创造一种与此完全对立的工作空间：一种让人精力充沛的环境。给予团队一个精力充沛的空间意味着让团队中的每个人感觉到他或她拥有自治权。每个人都可以决定自己的工作方式，并且有一定的自由可以做出改变，不仅仅是修改代码，还包括项目的整个计划，以及项目运作方式。极限编程团队通过那些面向计划的实践来做到这些：通过周循环来避免把可以延后的决定拿到前面做；通过"丢车保帅"来增加可以移动到下一个循环的次要任务。这些实践能够让大家感觉到更强的自治权，因为它们在项目计划上给予了团队成员更大的灵活性。自治权也可以来自代码本身：通过避免反模式，以及构建一个容易修改的代码库，团队可以给自己在项目后期留出更大的选择空间。

在第 3 章中，我们学习了*稳定步调*敏捷原则：敏捷过程倡导可持续开发。赞助商、开发人员和用户要能够共同、长期维持其步调，稳定向前。你知道为什么勇气这一极限编程原则是达到稳定步调的前提条件吗？

很多极限编程的实践者把"精力充沛的工作""40 小时工作周"和"稳定步调"这几个词作为同义词使用，这也是极限编程团队之所以通过设置合理的工作时间来创建一个可以让其精力充沛地工作的环境的原因。40 小时工作周其实颇有渊源，历史上，劳工组织曾经提

注 8：软件开发是脑力劳动对你来说应该不奇怪吧？如果不信，你可以自己证明一下。回忆一下你曾经花费好几个小时才写出来的一段代码。试着把它重新敲一遍。怎么样，这回快多了吧？把代码敲出来（也就是编程体力劳动的一面）显然不是最耗费时间的。

注 9：关于开发人员如何达成并保持流畅的工作状态，Tom DeMarco 和 Timothy Lister 在《人件（原书第 3 版）》一书中有很多精彩的论述。

出过“8 小时劳动、8 小时娱乐、8 小时休息”[10]的口号。到了 20 世纪 50 年代，无数生产效率方面的研究和图表显示，生产力在每周工作时间超过 40 小时之后就开始下降，因此很多产业也接受了这一原则。极限编程团队清楚，当它确定了一个稳定的步调并允许团队中的每个人享受工作之外的时间时，产品的质量也会随着成员的幸福感一起提升。

7.4.2　团队成员彼此信任并共同作出决定

> 人们需要一种“集体感”：我们属于某个集体。我们是一起的。我们支持彼此的工作、成长和学习。
>
> ——Kent Beck，《解析极限编程：拥抱变化（第 2 版）》

优秀团队的人们一起工作时，能够比各自单独工作所能完成的工作多得多。为什么会这样呢？因为一旦你把具有不同技能和视角的人聚集到一起，并给这些人提供一个鼓励开放式沟通和相互尊重的环境，就会促进创新。思想的碰撞会激发更多的好点子。

极限编程的*有凝聚力的团队*（whole team）这一实践是要让团队中的每个人都团结起来，成为一个整体。当遇到困难时，大家能够齐心协力去克服。当有一个可能影响项目方向的重要决定时，大家会共同作出这个决定。团队中的每个人都逐步学习信任团队中的其他人，并弄清楚哪些决定可以由个人作出，哪些决定需要经过集体的讨论。

在一个有凝聚力的团队中，每个团队成员都会参与到一系列的讨论中：什么功能对用户最有价值，团队下一步的工作内容是什么，以及如何进行软件开发。如果在编写代码以实现最大价值这一点上，每一位团队成员都获得充分的信任，那么程序员花额外的时间去编写不能达到上述目标的代码的风险就非常小了。

信任的另一面是要理解每个人都会犯错误。一旦团队的凝聚力起作用了，一个团队成员就不会害怕犯错误，因为他知道团队中的其他人能够理解错误是不可避免的，而且推进项目的唯一方法就是容许这些本就不可避免的错误发生，并且一起从这些错误中总结经验教训。

7.4.3　极限编程的设计、计划、团队和整体实践形成了一个带动创新的系统

一个在精力充沛的工作环境中工作的有凝聚力的团队，与一个在压抑的工作环境中工作的不团结的团队相比，其设计出的软件将会更好。创新也许听起来高不可攀，可实际上它就是一个搞笑的极限编程团队的日常工作。经常能够进入状态并且在鼓励渗透式沟通的高信息量工作空间中工作的开发人员，常常发现他们会互相激发对方的创意。同时，增量式的软件设计给予了每个开发人员编写新代码而不受束缚的自由，这就像给画家一个空白的画板。团队的好习惯会帮助开发人员避免和修正反模式，这样，当代码库增量式地成长时，团队就不会给未来的自己带来限制。

注 10：关于 40 小时工作周和 8 小时工作日，你可以查看维基百科（https://en.wikipedia.org/wiki/Eight-hour_day），了解更多信息。

当一个具备良好习惯的团队有一个信任它的经理时，更多好事情会发生。一个优秀的经理相信他的团队能够理解其所要实现的价值，并且会给团队创造一种氛围，让它能够把精力集中在创造最好的产品上，而不必去应付不切实际的工期。拥有这样一个经理的团队工作得更快，而且更有可能开发出高度可修改的、能够非常好地满足用户需求的产品。

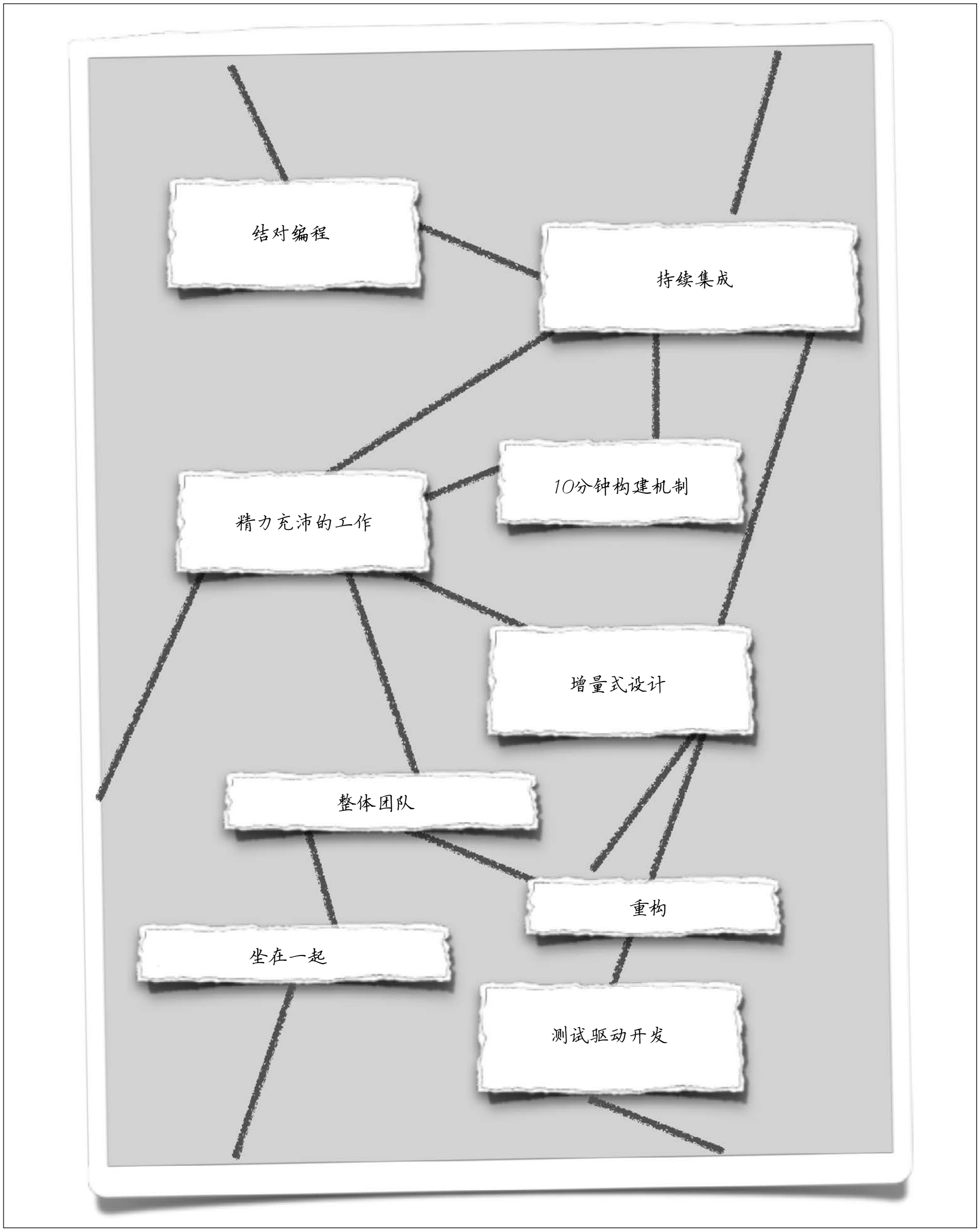


图 7-12：极限编程的实践之间相互关联，又相互促进

我们到第 7 章才介绍有关整体实践的内容，因为它们与其他极限编程主要实践相结合的时候才更有意义。一个不具备与极限编程的价值观和原则相兼容的思维方式的团队，是无法真正理解这些整体实践的。把各项实践分开来看，并一项一项地予以实施的团队可能会得到“聊胜于无”的效果，但是它不会看到这些实践对软件设计的深远影响。

在一个好的极限编程团队中，每个人都真正把这些原则和价值观内化了，所有的实践形成了一个互相促进的生态系统。测试先行可以帮助团队编写出小型、独立、解耦的代码，这使得代码问题和反模式更加容易被重构和剔除。团队成员坐在一起，并结对工作，这可以增进他们的凝聚力，进一步营造出更加专注、更加精力充沛的工作环境，而这种工作环境又使他们有足够的时间去思考软件的设计问题。他们持续不断地进行集成，所以当有人引入了一个 bug 并影响到软件的另一个部分时，被影响到的那部分代码的测试马上就无法通过了。于是，大家就可以迅速地修复代码的问题，避免这些问题被遗留在代码库中，这样，整个代码库也就更加干净了。持续集成之所以能够轻松做到，是因为它们的构建脚本运行时间不足 10 分钟，这同时也给团队带来更少的中断，从而让团队成员能够更容易地保持专注。

对于一个理解极限编程的开发人员来说，所有这些东西都非常简单。事实上，你会感觉开发软件就应该是这么开发，上面所说的一切都是自然而然发生的。

另一方面，如果不具备极限编程的思维方式，那么团队采用极限编程的方法时会遇到不少障碍。只顾着实施每一项极限编程的实践，看到每个人形式上都开始编写测试、结对进行编程、使用周循环或者使用持续集成了，团队就以为万事大吉了。这都是一些简单具体的东西，很容易做到。但是实施具体的技术细节与构建一个真正的“生态系统”是有很大距离的。对于后者的效果，有些团队可能只是听说过，却从来没有真正自己做到过，就像那些仅仅是实施了 Scrum 的具体技术细节的团队一样，从来都没有领略过那些惊人的成果和超高的生产力。

一个团队要怎样才能进入正确的思维方式中？它要如何实现从“单纯的技术实践”到“根本改变设计和编写软件的方法”的飞跃呢？

7.4.4 增量式设计与为了复用而设计

> Andrew: 这么说，为了复用而设计并不总是好事情？
>
> Auke: 这种做法的确不是很适合软件开发。开源软件的一般做法是，先为某个特定的应用场景编写一个软件，然后，其他人把你的软件拿过去，进行修改以适应他的需求。复用是这样进行的。而一旦你这么做了（即先使用、再复用），你就会发现各应用场景间的共同点，然后你再通过重构，把通用的功能提取出来。
>
> 这种模式你会经常见到。为了复用而设计过于僵化，很少奏效，也不适合软件开发。这也许是一个值得追求的目标，但是在可以预见的未来，是不可行的。“使用 – 使用 – 复用”这种模式更加适合软件开发。
>
> —— 对 Auke Jilderda 的采访，《团队之美》（第 8 章）

贯穿本书，我们一直在讨论一个团队如何通过引入某一方法的实践来逐渐进入正确的思维方式。一开始得到的可能只是“聊胜于无”的效果，但是随着团队逐渐熟悉这些实践并开始理解它们是如何结合在一起的，团队中每个人对软件开发的认识就逐渐开始发生转变。Scrum 是这样，极限编程也是一样。极限编程的一个关键就在于增量式设计。

Unix 操作系统的工具集是增量式设计的经典范例。该工具集的增量式设计原则使得数以千计的开发人员可以在很多年间为它贡献或大或小的代码片段。Unix 工具集包含很多独立开发出来的小组成单元[11]，大多数是由需要解决某个具体问题（而不是要构建一个大型的、一体化的操作系统）的开发人员写出来的。我们可以从 Unix 工具集这里学习到增量式开发是如何让大量不同的人（他们中很多人从未见过面）同时向一个大型的、可靠的、高质量的系统贡献代码，并且让该系统在数十年间持续不断地成长。

我们来举个例子。假如说你有一些很大的文本文件，里面保存着大量的数据，可能是某邮件列表中的邮件地址，或者操作系统的配置文件，或者是需要进行运算和处理的一些数字。你现在需要对这些文件进行快速格式转换（比如，提取出姓名和电话号码，或者对配置文件的某个部分做特定的修改，或者找出符合某种模式的数据），你会怎么做？

你大可以编写一个程序，从文件中读取数据，并进行相应的处理，然后生成输出。但是如果你对 Unix 工具集（像 `cat`、`ls`、`cut`、`awk` 和 `sed` 等）比较熟悉的话，就会知道这些东西就可以用来解决上述问题，所以你会写一个脚本（或者干脆就是一条终端命令）来读取文件中的数据，进行转换，然后把输出写入另外一个文件。

比方说你有一个很大的，用逗号分隔的地址簿文件 addr.txt。这个文件有 8 列：姓名、头衔、电子邮件、电话号码、通信地址、城市、州和邮政编码。你现在要找出 Minnesota 州所有人的姓名、头衔和电话号码。因为文件的前两列是姓名和头衔，电话号码是第四列，而州和邮编是最后两列，那么下面这条 Unix 命令即可将正确的输出写入 output.txt[12] 文件中。

```
egrep ",MN,[0-9]{5}([-][0-9]{4})?$" addr.txt | cut -d, -f1,2,4 > output.txt
```

Unix 工具集是由那些想要更容易地处理地址簿的人编写的吗？显然不是。Unix 工具集的哲学是“简单”：每个工具的输出都可以作为另一个工具的输入，而且每个工具都只做一个简单直接的具体工作。`cut` 这个工具的作用是：读取输入的每一行，把某些特定的列提取并输出出来。`grep` 的作用则是检查输入的每一行，并只输出那些符合某种模式的行。这两个工具的功能貌似简单，但是它们（尤其是当与其他 Unix 工具组合起来时）几乎可以完成无限种任务。比如，一个系统管理员可能把它们与 `find`（在文件系统中搜索文件）或者 `last`（告诉你最近都有哪些用户登录过当前主机）相结合，去完成很多复杂的系统管理任务。

注 11：你可以在维基百科页面（https://en.wikipedia.org/wiki/List_of_Unix_commands）查看完整的 Unix 工具列表，该页面加入维基百科的时间为 2014 年 7 月 27 日。

注 12：你是否注意到了我们的例子中有很多未考虑的边界情况？比如它假定所有的地址都是美国地址；它没有考虑表头行和数据中的逗号。如果你注意到了，先不要过度关注这些边界情况；这只是一个示例而已。

Unix 系统之所以能成为互联网服务和商务系统最流行的操作系统，这些工具功不可没。多年来，形成了一种通用的系统工具设计，同时也在那些每天使用这些工具的人们中间形成了一种系统管理员文化。随着时间的推移，新工具和新用法不断演化出来。例如，20 世纪 80 年代到 90 年代，文件压缩变得更加普及。1992 年把 `gzip` 工具加入 Unix 工具集时，有着一个非常明确的把它加入进来的模式。在写下第一行代码之前，数据的输入输出（通过管道）、程序的运行方式（通过命令行）、甚至文档应该放在哪（手册页）这些问题都十分清楚。该工具集可以很容易地进行扩展，而且由于所有的工具都彼此独立，新增压缩和解压缩工具可以不必修改系统的其他部分。

在过去 40 年间，Unix 工具集一直以这种方式在扩展。数以千计的开发人员贡献了各自编写的工具，或者对已有的工具进行改进。Unix 系统以一种自然、有机的方式不断成长。伴随着它的成长，也完成了一种文化和知识的积淀。这一切使那些使用 Unix 的人们能够完成越来越多的复杂任务，这也反过来帮助他们找到系统新的成长点。

7.4.5　简化单元交互，系统实现增量式成长

Unix 工具都遵循一种非常严格的输入输出模式。前面那条命令中的“|”这个字符叫作“管道”，它的作用是把一个工具的输出送给另一个工具作为输入。“<”把文件作为输入，而“>”则把输出写入文件。标准 Unix 工具会生成与这些管道相兼容的输出，这属于每一个单元都需要遵守的*协定*（contract）的一部分。Unix 工具之间的协定很简单，这使得扩展整个 Unix 系统非常容易。只要每个新增的工具都遵守相同的协定，它就能与其他工具保持兼容。

因此，帮助团队通过增量式设计来构建系统的一个关键在于在系统的各个模块之间建立一套简单的协定。各个模块如何相互沟通？是否要相互传递消息？是否要调用函数或方法？是否要访问网络服务？模块间的沟通机制越是简单一致，新增一个模块就越容易。

那么，怎样做到让协定简单呢？这与怎样让模块做得简单一样：到最后责任时刻再决定模块间应如何沟通。我们已经有一样工具可以帮助做到这一点，那就是测试先行编程。通过首先编写单元测试，开发人员必须在该模块还没有开发出来之前就开始使用它。她可以使用相同的单元测试工具或框架来编写简单的*集成测试*，即测试多个模块之间如何交互，她得在编写实际的交互代码之前就写好测试。举个例子，如果一个模块的输出会被传递给另外一个模块作为输入，而后一个模块的输出又需要作为第三个模块的输入，开发人员就需要编写一个测试来模拟这种交互，也就是这三个模块是如何集成到一起的。

这种做法有助于简化沟通协定的原因是，你可以很快地发现你规定的协定是否过于复杂。如果协定简单，那么这种集成测试应该很容易编写。但是如果协定复杂，集成测试就很容易变成一个负担：开发人员会发现她需要初始化很多完全不同且不相关的对象，对数据格式做各种转换，而且不得不经过很多个环节才能完成模块间的沟通。与单元测试一样，首先编写集成测试可以在这些问题进入代码库之前就将其避免掉。

7.4.6　优秀的设计源自简单的交互

有些时候，简单的元素组合起来，可以形成非常复杂的系统。这种现象在自然界中很普

遍：单个蚂蚁的行为非常简单，但是一个蚁群则因为单个蚂蚁之间的交互而产生了非常复杂的行为；蚁群作为一个整体，产生了一加一大于二的效应。

如果有一个系统，它的行为似乎是从各个模块之间的交互中自然而然地呈现出来的，而不是源自某个单一的模块，我们就把这个系统的设计称为*呈现式设计*（emergent design）。使用呈现式设计的系统几乎总是由小型的、独立的、解耦的模块构成的（如 Unix 工具集、蚁群）。这些模块可以通过组合来完成复杂的任务，而系统的行为和功能不仅来自每个单独的模块，更来自模块间的交互。

在这种设计中，很少见到模块间一层一层地互相调用；相反，模块间通过消息、队列或者其他去中心化的方式沟通。在这一点上，又要提到 Unix 工具集这个非常好的例子[13]。数据可以通过管道从一个工具流向另一个，这让多个工具的串联变得很容易。这使得对这些工具的使用就像一条流水线一样：调用第一个工具，然后调用第二个，再调用第三个，等等。工具之间也可以存在更加复杂的交互，但是它们都遵守一种简单、扁平的调用结构（相对于较深的、嵌套的复杂调用层次结构）。

当一个系统被设计成这样时（即简单的模块之间通过简单的方式互动），一些有趣的事情就会发生。系统的行为似乎是从整个系统中产生出来的，而不是从某个具体的模块。复杂的行为可能根本就没有一个单一的源头。就像蚁群一样，蚁群的行为似乎并不是源于任何一只单独的蚂蚁。团队会非常真实明显地感觉到系统的每一部分都很简单，而且系统的行为通过各个模块的交互而自然形成。

对于花费了多年时间开发 Unix 工具集的人们来说，这种感觉是非常熟悉的。从我们上面提到的地址簿的简单例子中，你能够看到一个非常基本的模块组合进而形成某些行为的例子：`cut` 只不过是从文本行中提取某些字符，而 `grep` 也仅需要关心如何进行模式匹配，但是它们组合起来，就可以处理地址簿，完全不需要告诉系统“地址”是什么以及该如何处理它。

顶尖的极限编程团队会很自然地使用呈现式设计来开发软件。这要从简单开始：每个模块都是为一个特定的用途而设计的。测试先行编程可以保证每个模块都保持简单，并且只有一种用途。开发人员首先针对该用途编写模块的测试代码，然后再编写功能代码，当所有测试都通过时，就结束编码，这样就不会增加多余的功能，也就没有冗余的代码。模块中的所有代码都是不可或缺的。

极限编程团队会避免过深的调用栈（一个模块调用第二个，第二个再调用第三个，如此往复）。程序的调用结构都比较扁平，这就减少了模块间的相互依赖。模块间的交互尽量简单：一个模块需要数据时，它就从另一个模块那里获取数据，或者更好的做法是，通过一个消息队列来获取，这样它甚至不需要知道输入数据是从哪里来的。为保持系统简单，极限编程团队会避免涉及很多模块的、多层次的、复杂的互动（比如 Unix 工具集，它就只是把数据通过管道一个一个向下传递）。

当团队中的人们发现系统有些地方需要修改时，就会发展出新的设计。要为系统增加新功

注 13：相互解耦的单元间通过简单的沟通而导致整个系统的复杂行为，蚁群也是一个很好的例子，只不过对蚁群来说，沟通是通过信息激素进行的。

能，或者修改原有功能，他们需要修改单个模块，或者修改模块间的交互。不过，因为模块之间是解耦的，而且他们的交互很简单，所以改动不会波及整个系统。如果发现了代码异味，比如一个程序员发现自己在做枪伤手术，或者遇到了乱麻式代码或半成品对象等问题，那么这就是一个警示信号，说明系统引入了不必要的复杂度，于是，发现问题的人知道（更重要的是，他确实认为）值得花时间去做重构，从而简化模块。

用呈现式设计构建的系统可以不断生长，同时保持其容易修改和维护的特点。善于使用极限编程的团队有一系列保持代码库简单的工具，其工作方式鼓励它不断地监控代码的复杂性，并随时通过重构进行简化。这种工作方式又反过来让团队能够更加顺畅地修改系统，进而促进设计的衍生。这种良性循环会产出简单、稳定、高质量的代码，于是，开发团队也会发现用这种方法来开发、维护软件更加快速。当系统需要进行维护时（修复 bug 或者改变功能），很少需要做涉及很多不同模块的、大幅度整体性改动。

同时，由于团队成员知道系统的灵活性，也知道他们能够进行哪些扩展，不能进行哪些扩展，他们就更善于在最后责任时刻做决定。他们会发现，作为一个团队，他们逐渐对将来可以做哪些具体的设计决策有了更清楚的认识，这又反过来帮助现有的代码保持简单。整个团队更加放松，成员间能够更好地协作。他们快速地产出高质量的代码，而且，当他们需要对代码做修改时，这种修改可以很容易，而且不会引入 bug。

换句话说，保持代码简单，使得开发团队可以拥抱变化。这正是极限编程的关键所在。

故事：有一个正在开发虚拟篮球网站的团队

- Justin——一位开发人员
- Danielle——另一位开发人员
- Bridget——团队的项目经理

7.5　第5幕：最终得分

Danielle 和 Justin 正在结对编程，这时候 Danielle 忽然意识到了什么。“你注意到没有？这是咱们俩三个多月以来第一次结对编程。”

Danielle 第一次跟大家提起极限编程已经是一年前的事情了。他们从那以后一直保持每周发布，不过上个星期那个周期比较特别。公司宣布要在一个大型电视网络上进行一次推广宣传，而开发团队刚刚把相关的代码部署到生产环境中。Danielle 和 Justin 正在对前一次迭代进行回顾，他们一直是这么做的。

Justin 说：“是啊。咱俩过去一直结对，不过我轮换了这么多结对伙伴，我觉得我跟团队中的每个人都一起工作过。”

Danielle 问到：“你上次加班是什么时候？”

Justin 想了想，说：“你猜怎么着？我查了一下手机上的道歉短信。那是很久以前的事了。”

“我觉得这是因为我们正在进步，”Danielle 说，“反正我知道我现在写出来的代码比半年前的要简单很多。”

“没错。我第一次了解到测试驱动开发时，感觉它特别理论化。现在一切都说得通了，”Justin 说，“我们以前常因为这个跟整个团队争论，试着说服大家值得花时间去这么做。现在我已经记不起上次争论去说服某人是什么时候了。”

Danielle 说：“说来也许好笑，我没觉得自己做事情的方法有什么不同。”

Bridget 刚才正好路过，于是停下来听 Danielle 和 Justin 的讨论。“我感觉起来有很大不同，”Bridget 说。Justin 和 Danielle 抬起头来。他俩刚才没有注意到 Bridget 站在一边。“我过去的工作主要是想办法让产品经理给我们更多的时间。那时候我们总是延期，而且他们要求的所有东西好像都要花费好几个月才能完成。”Bridget 说。

Justin 问：“现在不是这样了吗？”

“完全不是！我以前一直不断地说‘不行’，现在，我更常说‘可以’，哪怕他们要求的东西听起来很复杂。你们让我的工作轻松了很多。”Bridget 说。

“我猜这也是为什么你不再对我们大呼小叫的原因吧，”Danielle 说，“我猜这也是我们不需要周末加班的原因。”

Bridget 看了看 Justin 和 Danielle，突然说：“等会儿，这么说你们还可以再多干一些活喽？”

Justin 认为她在开玩笑。Danielle 则不这么认为。她说：“我觉得我们还在学习极限编程的路上。”

要点回顾

- 开发人员不断地对代码进行**重构**，去除代码异味、改进代码结构，但不改变代码的行为。
- 通过毫不犹豫的重构，极限编程团队避免陷入**技术债务**（即已知但尚未修复的问题）的泥潭。
- **丢车保帅**的一个很好的用处是，可以通过它来保证每个周循环都有足够的时间进行重构和偿还技术债务。
- 持续集成帮助团队及早发现集成问题，此时这些问题还较容易解决。
- 在重构、偿还技术债务，以及修复代码问题后，代码库变得更加容易修改。
- 极限编程团队通过**增量式设计**来写出由小型、独立模块组成的松耦合代码。
- 开发团队增量式地构建那些当前问题所急需的功能，并且相信它自己能够在最后责任时刻作出正确的决定。
- **精力充沛地工作**让极限编程团队创造了这样一种环境：团队中的每个人都能够以稳定的节奏进行工作，并且有足够的时间去做应该做的事。
- 极限编程团队的成员互相信赖，并且认为团队是一个整体，具有团队凝聚力。

常见问题

增量式设计听起来很理论化。真有人这样开发过软件吗？

当人们说某个东西“理论化”的时候，他们真正的潜台词是，“我没尝试过这个，而且那看起来很难。”

极限编程中没有什么是理论化的东西。它的每一个实践都是一项具体的技能，需要开发人员去学习、练习，并逐渐提高。他们在这些方面做得越好，就越能够把这些实践变成习惯，也就越能够写出更好的代码。价值观和原则都是真实、实用的思想，能够帮助你和你的团队提升技能，无论是个人的还是集体的。

这也是为什么极限编程是需要练习的。就像你不能简单地买个吉他，再买本《吉他速成》，然后明天就变成一个伟大的音乐家一样，你也不能一开始学习极限编程就期待自己能做得非常好。但是，如果你能坚持练习，你会有进步的。这也像学习音乐一样，如果你不断地犯相同的编程、设计和计划上的错误，你是没法进步的。

每次我把技术债务作为丢车保帅策略的一部分，都没能真正地去做它们。我如何才能保证技术债务能够被及时偿还？

如果你发现你的团队很少去做那些丢车保帅策略里的车，那说明你把太多的东西放进一个周循环了。那些加入周循环的次要任务并不是一些可做可不做的东西。它们只是一种为了保证团队能够按时交付确实完成的软件的保险措施。它们依然是重要的，而且大多数时候是应该被完成的。如果没有，那你应该在周循环计划中减少一些工作内容。

话说回来，如果说只是技术债务这类任务没有完成，那么你的团队可能对于技术债务存在一些认识上的误区。事实上，很多团队都经常陷入这一误区。这些团队会找出技术债务相关的任务，不过，它们不是尽快地修复它们，而是用卡片、表格、票证、或其他跟踪机制把它们放进积压工作表里面。技术债务被发现后，会被赋予比开发新功能更低的优先级。不知怎么的，这些技术债务最终都没有被修复，好像仅仅是承认它们的存在并把它们列入将来的修复计划就已经足够了。

一个不去修复技术债务的团队把它当作是开发过程的副产品。对这样的团队来说，交付新功能更加重要（哪怕需要用些取巧或山寨的方法），这样才能保证项目进度向前推进。

其实，这种态度并没有什么错，交付有价值的软件确实比发明那些漂亮的技术解决方案更重要。极限编程团队之所以不断地修复技术债务，是因为这么做能够让未来持续不断地交付有价值的软件变得更容易。

如果你是抱有这样一种心态的程序员，在编写新代码的同时还会修复技术债务，不断地进行重构，并且把这当作一件重要而紧迫的任务来做。团队中的每个人都知道尽快修复技术债务是值得花时间去做的事情，因为这能够让代码库更加容易修改（就像每月付清你的信用卡欠款是值得的，不付也不是不行，但是日积月累，欠款会最终变得无法偿

还）。这就是为什么极限编程团队的成员每周会给自己足够的时间（而且真正感觉他们有足够的时间）去偿还技术债务。

如果你发现你的技术债务正在累积，那么应该跟你的团队讨论一下这个问题。试着找出技术债务没有被及时解决的原因，是不是因为有来自外部的开发新功能的压力？或者也许原因是你和你的队友并不把它当作一件眼下最重要的事情？或者也可能是你的团队会奖励那些交付漂亮新功能的开发人员，所以大家会感到解决技术债务这种问题像是修水管，不会得到承认和赏识。如果说添加新功能和偿还技术债务需要经过相同的努力，但是如果你能够因老板知道你开发了一个用户喜爱的新功能而得到加薪，那么降低技术债务相关任务的优先级就是一个理性的选择。不管原因是什么，开诚布公地彼此讨论都会有所帮助。

有些极限编程原则感觉不太靠谱。之所以把“自相似”放进去，是不是因为它听起来很唬人，或者因为它是 20 世纪 90 年代人们都在讨论自相似、分形以及混沌理论时被提出来的？

绝对不是。如果你没有见过一条原则被有效地运用从而给团队带来改进，那么很容易会对它不屑一顾。但是我们所说到的每一条原则都确实能够帮助你更好地理解极限编程。自相似这一原则也不例外。

自相似的意思是一个相同的模式以不同的规模（或大或小）出现多次。优秀的软件设计中存在着大量的自相似。组成可靠、易用的软件的各个层次往往也是可靠、易用的；组成这些层次的那些模块往往也是可靠、易用的。要是你进入其中一个模块看看它的代码，你猜你会看到什么？没错：可靠、易于理解的代码。

在精力充沛的工作中也有自相似性：每个单独的开发人员都具备了“状态”意识时，团队才能够进入状态。计划中也有自相似性：一个开发人员先对测试进行了计划，然后编写代码；一个团队先对各项功能进行了计划，然后开始一个周循环；团队还会计划各项主题，然后开始季度循环。

那么是不是说你需要在工作的每个方面都尝试发现自相似性呢？当然不是，它不过是一项原则。但是它能够帮助你注意到一些规律，这些规律可以让你对自己的项目工作有一个更好的了解。其他的那些原则也是一样的作用。所以，不要忽视它们，它们是帮助你理解的有价值的工具。如果它们看起来有些多余，那么试想一下它们怎么能够应用到你的工作中。这会让你在采用极限编程的路上走得更远。

我看网上说测试驱动开发已死。是真的吗？

测试驱动开发是一个工具。说测试驱动开发已死，就好比说螺丝刀已死一样。很多团队每天都在用测试驱动开发，并且获得了很多成功。只要有人还在使用测试驱动开发，它就不但活着，而且活得很好。

但是，有关“测试驱动开发是不是已经死了？”的讨论还有更深层次的含义。人们（尤其是有很多敏捷和极限编程经验的人）提出这个问题的一个原因是他们发现有些团队不仅仅把测试驱动开发作为一个工具，而且把它视为一种正统。他们把测试本身视为一种终极目标，而不是通往“构建更好的软件”这一目标的一种手段。他们会陷入框架陷阱中，花费大量的时间去开发大规模的、复杂的测试框架。测试驱动编程本该是帮

助团队保持代码库简单易懂的；一旦它导致新增了更多的复杂性，那么肯定是出了什么问题了。[14]

其他声称测试驱动开发已死的人则是因为他们发现有些极限编程实践（尤其是测试驱动开发和结对编程）似乎催生了很多情绪化的抵制，主要来自那些不太熟悉它们的团队。

极限编程团队的目标并不是编写测试，就像采用瀑布式开发过程的团队的目标并不是编写规格文档。最终的目的都是构建出可工作的软件，而 TDD 和规格文档都是为了达到这一目的的手段而已。

还有一点可能会帮助你更好地理解这个争议以及它背后那些观念。在第 6 章和第 7 章中，我们通篇把测试驱动开发和测试先行编程作为可以互换的两个术语使用。这种用法是很常见的，而且多年来它们就一直被认为是同义词，而且我们也觉得这是一种合理的简化，有助于你理解这些概念。但是有些人认为测试先行编程仅仅指写代码之前先写测试，而测试驱动开发则指的是更广泛意义上的设计方法。

是不是说要写出设计良好的代码必须要先写测试呢？当然不是，很多团队没有使用 TDD 一样写出了非常优秀的代码。但是 TDD 是不是编写设计良好的代码的一种有价值的工具呢？毫无疑问是。如果从未尝试过测试驱动开发，那么你可以很容易地找出很多想象出来的反面理由（“我更喜欢把我要做的东西先粗略地写个大概，这样才能看出来各部分之间是不是有冲突，否则，我根本就没有可供测试的东西，所以，就 TDD 来说，我显然得作出一些妥协！”）。可是事实是，很多团队都是完全按照 TDD 的规定执行的[15]，而且当你认真地去执行时，很多想象中的那些问题就消失了。

其实，当初引起热议（“TDD 是否已死？”）的文章值得一看。还有，更要看一看 Kent Beck 发表在 Facebook 上的回复[16]，该回复很好地解释了 TDD 怎样帮助你解决编程的问题以及帮助你做出更好的软件设计。下面是该回复的详细内容。

> **抛弃 TDD**
>
> 2014 年 4 月 29 日，上午 11:10
>
> David Heinemeier Hansen（DHH）把 TDD 扫进了历史的垃圾堆（http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html）。我挺伤感的，并不是因为当初是我把它从历史的垃圾堆里给救回来的，而是因为我现在得找到一些新的方法来帮助我解决编程中遇到的下列问题了。

注 14：你是否发现自己在编写复杂的单元测试？你是否刻意地去追求 100% 的测试覆盖率，很可能是通过编写深思熟虑的测试用例或者编写非常复杂的模拟对象？你的单元测试是不是难以修改？有没有一些测试用例被人注释掉了以便能够通过测试，因为该测试太复杂难懂，没法修改？如果你对上面任何一个问题的回答是肯定的，你的单元测试很可能正在使你的代码库变得更加复杂，而不是更加简单。

注 15：包括本书作者。我们编码时总是习惯使用测试驱动编程。

注 16：有一小部分人可能看不出 Kent Beck 的 Facebook 帖子中的反讽意味。非得解释一个笑话的话，这个笑话就不好笑了，不过为了避免误解，我们还是解释一下：Kent 的意思是如果 TDD 真的死了的话，他就得找到一个替代 TDD 的方法去做那些 TDD 原来能够帮助他做到的那些有价值的事情。

- 过度工程问题。我总是禁不住会把一些知道自己将来会用到的功能加入软件。让一个失败的测试由红转绿（还有一系列未来的测试）可以帮助我只实现必要的功能。现在我需要一种新的方法保持专注。
- API 反馈。我需要找一个新的方法来获取关于我的 API 设计的反馈。
- 逻辑错误。我需要找一个新的方法来发现那些我常犯的逻辑错误。
- 文档。我需要找一个新的方法来告诉别人我的 API 应该如何使用，以及记录我在开发过程中的思路。
- 应接不暇。我肯定会怀念使用 TDD 的时候，不会感到应接不暇，即便我无法想象一个整体的实现，我几乎总是能够写出一个测试来。现在我得找到一个新的方法，帮助我走出万里长征的下一步。
- 把接口和实现分离开来的思维方式。我容易用对实现的臆测污染 API 的设计。现在我需要找到一个新方法来分离这两个层面的思考，这个方法还得能够实现这二者之间的快速反馈。
- 伙伴间的约定。我需要找到一个新方法来准确地告诉我的编程伙伴我在解决什么问题。
- 心理上的焦虑。可能我最会怀念的是 TDD 能够瞬间回答“一切都没问题吗？”这个问题。

> 我肯定自己能够找到上述问题的其他解决方案。痛苦会随时光的流逝而渐渐淡去。再见了 TDD，我的老朋友。
>
> ——Kent Beck，Facebook 页面（https://www.facebook.com/notes/kent-beck/rip-tdd/750 840194948847）

那么最终的结论是什么呢？有些团队尝试了测试驱动开发并发现它非常有用。另外一些团队则发现它很难操作。要想知道它对你是否有用，最好的方法就是试一试。如果你真的要尝试（我们建议你试一试），那么得到好结果的一个方法是当你构建你的单元测试时，小心不要陷入框架陷阱。

上面 Kent Beck 的回复针对的是 David Heinemeier Hansson（Ruby 社区通常叫他 DHH）的一篇文章。DHH 是 Ruby on Rails 的创始人，同时也是敏捷社区的一个重要的思想者。他的那篇文章标题是“TDD 已死，测试万岁”[17]DHH 的这篇文章中非常犀利地指出了陷入框架陷阱的单元测试的以下特征。

> 测试先行导致了过度复杂的、充满中间对象和间接调用的蜘蛛网，其目的是要避免去做那些所谓“慢”的事情，比如访问数据库，或者文件操作，或者同构浏览器来测试整个系统。这形成了一个充斥着服务对象、命令模式以及更糟糕的东西的茂密丛林。
>
> 我很少使用传统意义上的单元测试。在那种单元测试中，所有的依赖关系都被模拟了，上千个测试可以在几秒钟之内跑完。

注 17：David Heinemeier Hansen，“TDD 已死，测试万岁”，详见 http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html（2014 年 4 月 23 日）

上面的这段描述是不是很耳熟？你应该感到耳熟，因为它跟你在本章中学到的那些代码异味很相似。单元测试也是代码，像其他代码一样，它也容易受到那些复杂性问题的影响。如果你发现自己编写的单元测试非常复杂，那么你可能已经陷入 DHH 指出的那个误区。

DHH 同时也指出了 TDD 的下面这个重要的特性。

> 开始可不是这样的。当我刚接触测试驱动开发时，它就像是一张通往更美好软件开发世界的谦恭邀请。它让你从完全不测试到开始使用测试。它让我见识到了经过良好测试的代码的稳定性，并且它解放了那些需要修改软件的人。
>
> 测试先行是一系列非常好的训练工具，它们教会了我如何从一个更深刻的层次去思考测试这个问题，但是我也很快抛弃了其中的一些东西。

这是一个尝试 TDD 的很好理由。它确实能够让你从更深的层次去思考测试，正如它也能做到 Kent Beck 的 Facebook 帖子中所列出的那些事情一样。TDD 是不是做到这些的唯一方法？不是。但是它很有效，而且这也正是它成为极限编程的一个重要部分的原因。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 如果你是一位开发人员，试着重构你的代码。你可能已经在使用具有内建重构功能的 IDE 了。你能否稍微修改一下你正在写的代码，将其简化，但不改变它的行为？
- 到 WikiWikiWeb（http://www.c2.com/cgi/wiki?CodeSmell）的代码异味页面，通读该页。你能在你的代码中找到这些异味吗？如果找不到，再好好找找。
- 你的代码是否能很容易地从代码仓库检出并构建？如果需要很多步骤，试试看能否通过一个构建脚本将其自动化。长期来讲，简化检出和构建过程的每一个努力都会让你的生活更美好。
- 你尝试过测试驱动开发了吗？没有？快试试吧。拿一个你要做的用户故事或者功能特性。开始写代码之前，为某一个单元写一两个测试。你不用急着写出一个完整的测试套件，就写一两个单个的测试，然后编写代码让它们通过。

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 关于简化、增量式设计以及其他极限编程的整体实践：《解析极限编程：拥抱变化（第 2 版）》，Kent Beck 和 Cynthia Andres 著
- 关于重构：《重构 : 改善既有代码的设计》，Martin Fowler、Kent Beck、John Brant 和 William Opdyke 著。
- 工作流和如何创造一个精力充沛的工作环境：《人件（原书第 3 版）》，Tom DeMarco 和 Tim Lister 著。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 你的团队是否陷入过框架陷阱的误区？是否曾尝试去解决大型的抽象问题而不是小型的具体问题？帮助团队认识到这种思维是复杂的。把思维方式从复杂转向简单是帮助一个团队适应极限编程的最重要方法之一。
- 与团队讨论 YAGNI（你不会需要它的）。帮助团队找到至少一段用不到的代码。很多团队开发了完全没有被用户提出过的功能，所以也完全没有人用。你的团队是否做过这样的事？
- 留意团队成员对某项实践（如测试驱动开发或者结对编程）的不适应，帮助他们克服这种不适应。避免针对这些实践的优点的冗长哲学辩论，把精力放在寻找这些实践能够帮助该团队解决的问题上。经验是克服防御性和不适应的最佳方法，有时候只要让一个团队成员花一两个小时自己亲手试试就解决了。
- 有些实践（尤其是测试驱动开发）会吸引某一个对它高度感兴趣的人，但是他的这种激情可能在团队其他成员看来是过度狂热的。帮助这样的人意识到他应该慢慢来，而且团队中的其他成员尚未达到跟他相当的程度。耐心是最难传授的，所以，你自己得先做到有耐心。

第 8 章

精益、消除浪费和着眼全局

精益是一种思维方式，是一种世界观。

——Mary Poppendieck，Tom Poppendieck，

The Lean Mindset: Ask the Right Questions

到目前为止，你已经学习了 Scrum 和极限编程。这两种方法都提供了各自的具体实践，同时还提供了各自的价值观和原则，以帮助团队中每个人进入一种高效的思维方式。如果你每天参加每日例会，使用冲刺，并与产品所有者和 Scrum 主管一起工作，那你就是在使用 Scrum。对极限编程也是一样：如果你不断重构，使用测试驱动开发，持续集成并且进行增量式设计，你的团队就正在使用极限编程。

极限编程和 Scrum 有一个共同点：如果不理解它们的价值观和原则，那么你将仅仅止步于表面的做法，得到“聊胜于无”的结果。正如 Ken Schwaber 所指出的，如果你不理解集体承诺和自组织，你就没有理解 Scrum，而 Scrum 的价值观正是要帮助你理解这两点。同样，在极限编程中：不理解简化和精力充沛的工作这些价值观，你最后不过是把极限编程的实践当作一个清单，你的团队不会真正拥抱变化，而你最后得到的还是难以维护的复杂软件。

精益却不一样。不像 Scrum 和极限编程，精益并不包含一组实践。*精益是一种思维方式*，像 Scrum 和极限编程的思维方式一样，精益也有它自己的价值观和原则（精益的术语把它们称为“思维工具”）。精益的思维方式也常被称为*精益思维*（lean thinking）。“精益”这个词多年前曾经被用于制造业；21 世纪初由 Tom 和 Mary Poppendieck 引入软件开发领域。本书中的“精益”一词指的就是这种在软件开发领域中对于精益思想的引申。

在本章中，你将学习什么是精益，那些帮助你进入精益思维的价值观，以及精益的思维工具，它们可以帮助你找出和消除浪费，并且让你看到你用以开发软件的整个系统。

8.1 精益思维

精益是一种思维方式。这种说法挺有意思（给思维方式专门起一个名称），也很实在。

在本书前面部分我们已经看到，要有效地采用 Scrum，一个团队需要具备一种特定的思维方式。我们也看到，Scrum 的价值：投入、专注、开放、尊重和勇气帮助团队进入这种思维方式中。我们还看到，极限编程也有它的思维方式，极限编程团队也一样有简化、沟通、反馈、尊重和勇气等一系列价值观来帮助团队进入正确的思维方式。

所以，精益有它自己的一套价值观，并且要采用精益思维的团队需要从这套价值观开始也就没什么稀奇的了。

精益的价值观包括以下这些。

- *消除浪费*
  找出那些不能直接帮助你创造出有价值软件的工作，把它们从项目当中去掉。
- *增强学习*
  通过项目的反馈来改进你开发软件的方法。
- *尽可能延迟决定*
  每一个项目的重要决定都要等到你拥有了最大量信息的时候再做，也就是在最后责任时刻。
- *尽快交付*
  理解延期的代价，并且通过拉动式系统和队列来将这种代价最小化。
- *帮助团队成功*
  创建一个专注而高效的工作环境，创造一个由精力充沛的人组成的团队。
- *保证产品完善*
  软件应该符合用户直觉，并形成一个一致的整体。
- *着眼全局*
  全面理解项目中的工作——使用恰当的衡量指标来保证你了解全局，毫无遗漏。

每个价值观都有对应的“思维工具”帮助你把它们应用到你团队的现实中去。每个思维工具可以大致类比成极限编程的一条原则，它们的用法是完全一样的。随着我们讲解每一条价值观，我们会连带讲解其对应的思维工具，以及你应该如何使用它们来帮助你的团队进入精益思维模式。

8.1.1 你已经理解了很多精益价值观

> 一个组织更加珍视什么，它就最终得到什么。在我们把价值取向从过程转向人、从文档转向代码、从合约转向合作、从计划转向行动的过程中，敏捷宣言起到了很大的促进作用。
>
> ——Mary Poppendieck，Tom Poppendieck，《敏捷软件开发工具》

如果告诉你很多敏捷价值观和思维工具你其实已经了解过了，你会感到意外吗？你不该意外。精益是敏捷开发的一个重要部分。当 Poppendieck 夫妇把精益制造的一些思想应用到软件开发中时，他们从敏捷开发的其他组成部分（包括极限编程）中借鉴了一些东西。更重要的是，他们所用到的制造业中的思想，其实已经作为工程和质量管理领域的重要部分存在了几十年了。Ken Schwaber 在发明 Scrum 的时候就从这些质量管理的思想中借鉴了很多，我们在学习每日站立会议就是例行检查这一点时已经看到了。

由于本书涵盖了多种敏捷方法，所以我们已经讲过一些精益中的重要内容。我们在这里简要地回顾一下，之所以不用过多的篇幅来讲解是因为你对它们的了解已经比较深入了。它们仍然是精益思维的核心元素。

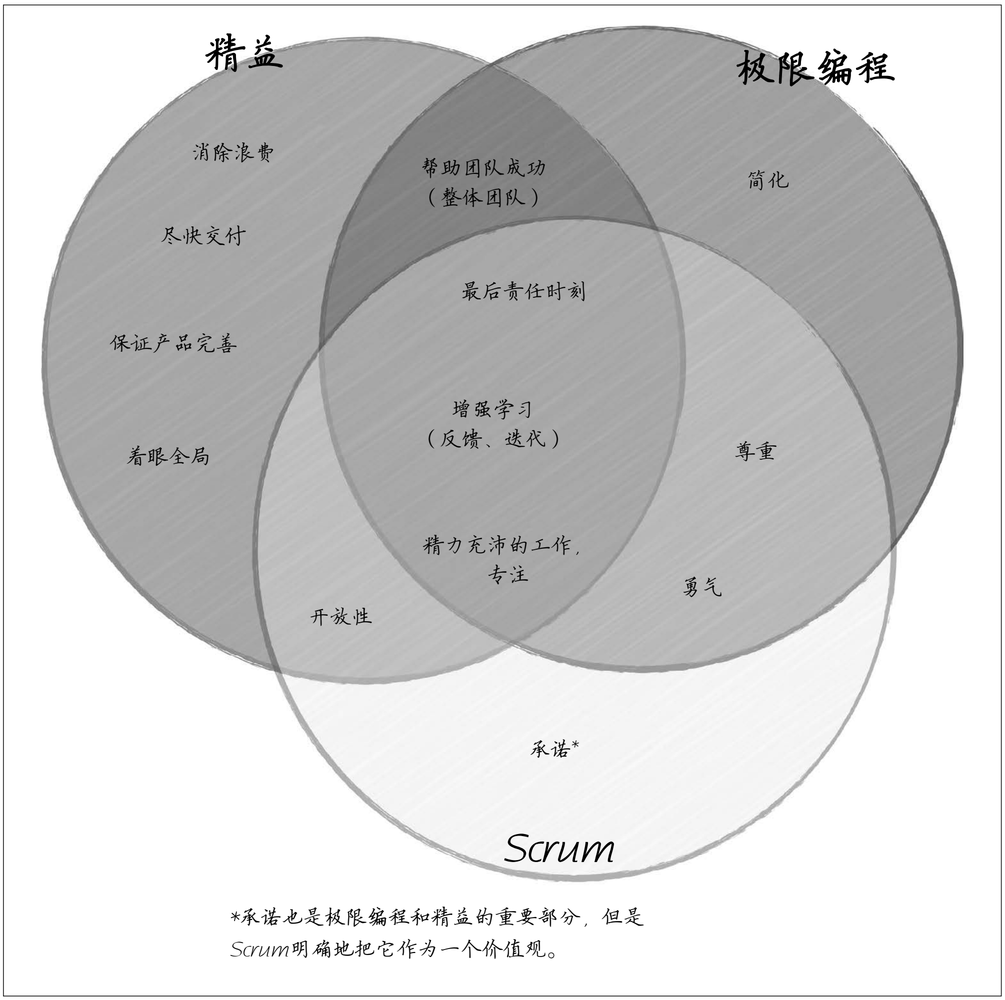


图 8-1：你对精益已经了解了很多，因为精益思维的价值观与 Scrum 和极限编程的价值观有很大的交集

再看一眼精益价值观。至少有一条应该引起你的注意：尽可能延迟决定。Scrum 和极限编程都严重依赖在最后责任时刻作出决定这一思想。而这与精益价值观是完全一样的内涵。Scrum 团队在计划中应用这一思想。极限编程团队也是，它同时也把这一思想应用到设计和编码中。精益的实践者也会这么做，事实上，这条价值观甚至有一个思维工具就叫“最后责任时刻”，其内涵与你已经学过的完全一样。

还有一个价值观你也已经学过了：增强学习（amplify learning），之所以说你已经学过了，是因为这一价值观的两个首要的思维工具就是反馈（feedback）和迭代（iteration）。它们的内涵与你在 Scrum 和极限编程中学到的完全相同。这一价值观还有另外两个思维工具：“同步”和“集合式开发”。同步与极限编程中的持续集成和共同所有权很类似。至于集合式开发，我们很快就会讲到。

最后，你还学过帮助团队成功（empower the team）这一价值观，以及它的思维工具自行决定（self-determination）、积极性（motivation）、领导能力（leadership）和专业知识（expertise）。这些思想与极限编程的整体团队和精力充沛的工作这两项实践背后的思想几乎是等同的，尤其是在避免加班这个问题上，因为加班对产品有着严重的负面影响。你在第 4 章中已经了解过 Scrum 团队也很重视这一点，这也是 Scrum 的专注（focus）价值观的一部分。一个团队，其成员能够控制自己的生活，这个团队才能开发出更好的软件。后面你会了解到，着眼全局的一个很重要的方面就是把项目相关的信息分享给每个人，包括经理，而这也恰恰是 Scrum 团队重视开放性（openness）的原因。

8.1.2　承诺、选择意识和集合式开发

下面这段话出自本书第 4 章（颇值得重读一遍）。

> 计划并不会让我们承诺。我们自己的承诺才是承诺：计划只不过是一个方便记录这些要承诺的事情的地方。只有人才能做到承诺，项目计划只是负责记录。当有人指着项目计划说已经作出了承诺的时候，这个人并不只是在说一张纸而已。人们关心的是写在这张纸上面的承诺。

这段话到底是什么意思？它的意思是，制订计划的时候，你首先作出了一个承诺，然后把它落实到书面上。

现在再回头看一下图 8-1。我们给承诺加了注解，特别说明了它是精益和极限编程的一部分，因为尽管只有 Scrum 把它单独列为一条价值观，但承诺对于极限编程和精益思维来说都是非常重要的一部分。不过精益更近一步地给承诺增加了一些细节：选择意识（options thinking），即意识到你所承诺的事情与你有权力（但没有义务）做的事情之间的差别；集合式开发（set-based development），即在项目开发过程中，让团队并行开发几个可供选择的版本，这样就可以有多个选项可供比较和选择。

1. Scrum团队承诺创造价值，但是在如何做到这一点上保留选择权

团队应该作出承诺[1]，不是吗？

注 1：关于选择意识，有一个很好的学习资源。这是一本名为 *Commitment* 的图像小说，作者是 Olav Maassen、Chris Matts 和 Chris Geary。感谢 David Anderson 把该书推荐给我们。

举例来说，当一个 Scrum 团队的成员被要求在几个月内交付某个具体的功能时，他不能直接告诉用户和经理："我们采用的是 Scrum，所以我不需要承诺超出本次冲刺以外的任何事情。两个月后再说吧。"相反，Scrum 团队有一个产品积压工作表，它可以通过这个积压工作表来与客户协作并理解哪些东西是有价值的。团队不仅承诺在本次冲刺结束时交付有价值的软件，而且还承诺在下一次、再下一次，以及接下来几个月间的每一次冲刺结束时都做到这一点。

将上述 Scrum 团队的做法与上令下行式的项目经理的做法比较一下：后者会制订一个非常详细的项目计划，把人力投入到未来几个月的具体任务上。当一个团队提前就交付某项功能作出了承诺，而且进行了周密的计划时，就给了每个人一种"一切尽在掌握"的幻觉。项目经理可以拿出分配给某个开发人员的一个具体任务，让他在四个星期后的周二上午 10：30 做这个任务；这会让每个人认为团队已经考虑了所有的选择并投入到那条最佳的路径上。问题是团队并未真正投入到某件事上。团队没有足够的信息做到真正的投入。事实上，在这种情况下，团队只知道一件事：计划上给这个程序员规定的四周后的周二上午 10：30 该做的任务，几乎肯定是错误的。[2]

因此，Scrum 团队不是对每个任务进行周密的计划并被该计划束缚，而是自我组织并共同投入到创造价值这一目标上。同时，团队不会承诺让某个开发人员在某个具体时间，如四周后的周二上午 10：30 执行某个具体任务，相反，团队会将决定留到未来的责任时刻（很可能是四个星期后的每日站立会议上）。

但是，Scrum 团队在冲刺开始之前是不会承诺交付某项具体的积压工作表条目的。而且即使冲刺已经开始了，如果某些条目不再有意义，产品所有者依然可以把它们从该次冲刺中移除出去。通过作出一个更小的承诺（创造价值），而不是承诺交付具体的功能，Scrum 团队把决定和承诺推迟到了一个更晚的责任时刻。

当冲刺开始的时候，冲刺积压工作表中的条目感觉像是团队所作出的承诺，因为团队成员在冲刺计划会上进行了讨论并把这些条目加入了任务板。可是，这些不能算是承诺。它们只是选择而已。之所以这么说，是因为产品所有者随时可以把它们拿掉，而且在冲刺临近结束的时候，如果团队发现无法交付"真正完成"的功能，就会把它推迟到下一次冲刺。这是 Scrum 之所以有效的一个原因：它有效地区分了真正的承诺（在每个规定了时限的冲刺结束时，交付有价值的、可工作的软件）和选择（在某个具体的日期交付某一项具体的功能）。

让人们去思考选择而不是承诺可能会很难，因为在谈及"承诺"的时候，其含义并不总是清晰的。没有人喜欢被强迫马上作出承诺。我们大部分人都经历过这样的场面：会议上老板要求某个团队成员给出具体的完成日期，后者不自在地在座位上挪了挪，给出了一个似是而非的承诺。这种情况的发生一般是因为老板或者项目经理感觉他们被团队搞得焦头烂额，因为团队做了承诺却没有做到。这时候他们的本能反应是对团队做微管理，要求个别

注 2：如果你有金融方面的背景，你会发现就选择意识这一点来说，与金融服务领域的情况有很多共通之处。交易员和投资组合管理员会买入或卖出期权（一种证券，它给予交易的某一方以这样一种权利：在某个特定的日期、以某一特定的价格买入或卖出某种股票或商品，如原油、小麦）。这使得他们可以使用这样一种策略，即买入某个股票或商品，但不必彻底占有它。换句话说，他们为选择留有余地。

团队成员就很多短期任务作出大量的承诺。这就制造了一种让开发人员害怕作出承诺的环境。

老板和经理不信任团队，团队反复违背自己的承诺，一个项目是如何走到这一步的呢？之所以会产生这样的结果，原因在于，尽管个人不愿意做承诺（尤其是在老板面前压力山大的情况下），但是团队却倾向于过度承诺。有时候可能是因为英雄主义，比如激情过剩的开发人员承诺了自己无法做到的东西。还有的时候，原因是承诺能够带来好处：今天先做个承诺，假如明天真发生了什么意外导致项目无法完工，道歉就是了。（"当初谁能想到会这样呢！"）在一个充斥着推诿责任和自我保护文化的团队里，这种"先承诺，再道歉"的做法尽管对交付软件没什么用，却经常是获得更高的薪水和抱住饭碗的最佳途径。

在这里我们应该再重复一下问题的核心：老板们常常会倾向于要求承诺，而团队则倾向于过度承诺。

2. 增量式设计、集合式开发，以及其他带来更多选择的方法

Scrum 任务板上的一个任务要经历三种状态：待处理、处理中和已完成。当一个任务处在待处理一栏的时候，它就依然是一个选项，而不是一个承诺。而且，即便一个任务已经在处理中，如果有必要，团队还是可以改变方向。在第 4 章中，我们讨论了一个 Scrum 团队不会花大量时间去跟踪和模拟任务之间的依赖关系，因为这些依赖关系的作用在于预测日程。相反，Scrum 团队每天都可以在每日站立会议上自由地往任务板上添加或者从任务板上移除任务。这些任务都只是选项，而不是承诺：它们没有截止时间，而且不存在会让整个项目无法按时完成的所谓的延期任务。这能够鼓励团队中的选择意识，因为它们只就一个目标做了承诺，而且可以随时基于新获取的信息改变具体的任务从而达成该目标。

这里是 Scrum 团队如何运用选择意识的一个例子。比方说在某次冲刺计划时，一个 Scrum 团队把某个故事分解成了四个任务，基于的假设是数据将存储在数据库中。于是就有了一个数据库设计任务，很可能得需要一个数据库管理员来做。冲刺进行了两周，一个开发人员发现，他们需要的数据在数据库中已经以另外一种格式存储起来了，这样一来，更合理的做法是创建一个对象或者服务，把已经存在的数据整理成系统其他部分可用的形式。对于该 Scrum 团队来说，这是一个好消息！团队可以在接下来的每日站立会议上把原任务移除，这样就可以腾出时间来做别的事情，比如新增一个积压工作表条目或清理一些技术债务。

相比之下，对于一个传统的瀑布式团队，发现一个任务应该由另外一个人来完成可能会很难处理：项目经理必须重新进行资源分配，因为现在不需要 DBA 来完成这个任务了，而这可能会影响到项目间的依赖关系，从而波及其他项目。如果受波及的项目计划包含了很多需要重新协商的事项，就会产生问题。这样一来，项目经理可能禁不住会要求团队干脆还是使用原来的方案。或者做原始设计的那个技术架构师可能会感觉本来说好了要用他的基于数据库的方案的，但现在开发团队却说了不算。

很多开发人员都体会过这种感觉，那些并不身处一线的人，比如项目经理或者技术架构师，要求一线的开发人员做一些不必要的，甚至是影响效率的技术妥协。从开发人员的角度看，开发团队被要求在软件的质量上做妥协。从项目经理或架构师的角度看，开发人员没能信守承诺。两方面都有理由把对方当作坏人。

真正的问题是，软件设计原本就不应该被定死，如果每个人都把软件设计看作是一个选项，而不是一个承诺，那么团队就能自由地开发最好的软件，而项目经理和架构师也不会因为他们的计划被改动了而感到自己被轻视了。

那么怎么才能让老板和团队成员都少做承诺，并把所谓的“最终”决定看成是选项呢？

极限编程给出了一个很好的答案：增量式设计。当一个团队构建简单的组件，而这些组件中的每一个都只做一件事时，团队就可以以多种不同的方式对这些组件进行组合。开发团队可以通过重构和测试驱动开发来保持这些组件尽量地相互独立。

换句话说，解耦的、独立的组件可以为我们提供更多选项。

有了新的想法时，或者发现了一个新需求时，如果组件过于庞大，并且有很多其他组件依赖它们，开发团队就得把大部分精力用在对代码进行大幅改动并进行枪伤手术上。但是极限编程团队因为使用了增量式设计，所以留有选择的余地。团队可以仅增加最少量的代码来满足新的需求，而且会继续重构并继续使用测试驱动开发。团队成员会尽量避免增加不必要的依赖，这就让他们继续做到为将来保留尽可能多的选项。

所以，极限编程和 Scrum 团队的选择意识是通过其工作计划和软件设计体现的。那么有没有什么方法可以直接为项目创造更多的选项呢？

当然有。这个方法就是所谓的*集合式开发*。开发团队会花时间对选项进行讨论，并且调整工作方式，从而给未来增加额外的选项。团队成员会做额外的工作来实现多个选项，并且相信这些额外的工作是值得的，因为这为他们提供了更多的信息，以便将来他们能够作出更好的决定。

假设一个团队面临一个业务问题，该问题有不止一个技术解决方案，比如，我们的 Scrum 团队在冲刺进行到一半的时候发现它不需要 DBA 来修改数据库了。如果开发团队并不确定基于对象的方案是否比基于数据库的方案更好呢？这在软件项目中很常见。你常常不知道哪个方案最好，除非你把它们都实现了，或者最起码得开始实现它们。

当开发人员解决困难的问题时，在真正开始做之前，他们并不完全知道要解决该问题到底涉及哪些东西。每个开发人员都很熟悉这样的场景：开始以为“简单”的问题，最后发现其实非常复杂，或者比想象的要难得多。这对软件团队来说可以说是一个基本常识。如果开发团队基于“问题很简单”这个假设作出了某些承诺，结果发现比想象中的要复杂得多，那么它只有两条路可选：要么说了不算，要么就交付粗糙、山寨的代码并积累技术债务。

集合式开发可以帮助我们的 Scrum 团队应对这种不知道那种方法更好的局面。在集合式开发中，开发团队不是从两个选项中选择一个（数据库方案、对象方案二选一），而是把实现两种方案的任务都加到任务板上，同时实现这二者。乍一看这可能感觉有些浪费，但从长期角度讲，这能够为团队节省很多精力。如果证实某一个方案确实更优，而另一个方案会导致粗糙的设计和大量的技术债务，那么同时实现二者是值得的，最起码它给了实现这两种方案的开发人员以时间去全面考虑他们的方案。这给了他们更多的信息去做更好的决定，而且最终做决定的责任时刻是在他们花时间解决问题之后。

另外一个常用的集合式开发的例子是 *A/B 测试*（A/B testing）。这个做法常见于开发用户界面和改善用户体验，而且这种做法在亚马逊、微软和很多其他公司中都取得了很大的成

功。所谓 A/B 测试，是指开发团队实现两个或多个解决方案，比如，一个网页界面的两种不同的页面布局或者决策路径（一个 A 布局，一个 B 布局）。然后开发团队随机地把这两种布局分配给测试用户（或者在某些项目中，直接给真实用户），并监控他们的使用率和成功率。企业普遍发现这种做法虽然需要花费更多的时间和精力来开发两个不同的版本，但是长期看是很值得的，因为他们可以采集一些指标来证明其中的某一个解决方案更加成功。不仅如此，较差的那个方案也能给团队提供经验，比如该方案的某些特性可以加入到最终产品中。

这些例子说明了团队如何在现实生活中使用集合式开发和选择思维。更重要的是，它们也说明了你已经学过的 Scrum 和极限编程的思想可以作为你学习精益和精益思维的一个出发点。

要点回顾

- **精益**或**精益思维**是一种思维方式，与其他敏捷思维方式一样，它也有一些价值观可以帮助你理解和适应它。
- 精益是一种思维方式，而不是一种方法。这也是为什么它没有可供你用到项目开发中的具体实践。
- 精益思维与更广义的敏捷开发有一些共同的价值观：“尽量推迟决定”（或者在最后责任时刻做决定）和“增强学习”（使用反馈和迭代）。
- **帮助团队成功**这一精益价值观与 Scrum 的专注和极限编程的精力充沛的工作十分相似。
- **选择思维**的含义是理解选项与承诺的区别，并且尽量给你项目的未来留出尽可能多的选项。
- 使用**集合式开发**的团队会同时探索多个选项并从中选出最好的一个，像 **A/B 测试**中在一个用户样本上测试多个用户界面选项并选择最佳的那些特性。

故事：有一个正在开发一个手机相机应用的团队
（团队所在的公司刚刚被某大型互联网集团公司收购）

- Catherine——一位开发人员
- Timothy——另一位开发人员
- Dan——老板

8.2 第1幕：还有一件事……

“大伙儿加油啊！鼓起劲儿来好好干！”

Catherine 简直烦透了老板的这句吆喝。她刚跟老板开完了会，而老板对团队在某个他要求加上的特性上的进展不满意。负责该特性的那两个开发人员遇到了一个问题，导致需要花费更长时间，可是老板却不愿给他们更多的时间。他倒没明确地说要是不能按时完成就得

面对后果，可是大家都清楚，延期肯定没有好果子吃。

去年 Catherine 可比现在爽多了，那时候她在一个仅有一个开发团队的小公司工作。大家都在开发同一个产品，一个给手机相机增加额外功能的应用。团队很小，但是大家都非常有创新意识，而且每个人都知道如何协同工作。

这个团队在业界也颇受关注。很自然地，一家大型互联网集团公司表示有兴趣收购这个团队所在的公司。Catherine 和她的伙伴们当时非常兴奋，因为他们听说了很多关于那家大公司的传言：宽松的工作环境、装满饮料的冰箱、灵活的工作时间以及其他福利。当交易最终完成时，他们都得到了高额的奖金（Catherine 自己还完了学生贷款！），然后整个团队搬到了市中心豪华的新办公室。

“咱们是怎么搞到今天这般田地的，Timothy？”在走出老板办公室的时候 Catherine 问道，“咱们做的事情本来很有意思啊。到底是怎么了？”

Timothy 是原团队的一名程序员，在团队里待的时间跟 Catherine 差不多长。“我不知道，”他说，“似乎所有事情都得花两倍的时间才能搞定。”

“我不介意加班，”Catherine 说，“可是我感觉不管怎么加班怎么努力，我们总是落后于计划。”

“是啊，”Timothy 说，“而且我们永远也不可能把进度赶上来。”

“嘿，Cathy！ Tim！你俩有时间吗？”

Catherine 和 Timothy 对视了一眼。“看来不妙。”Catherine 说。他们的老板 Dan 正叫他们两个人到他的办公室去。

“我刚跟一个负责我们公司社交网络站点的资深经理通了电话，有个好消息。”（莫名其妙，一切新需求都是“好消息”了。）“他有个想法，想把他们那边社交网络的数据与我们的相机应用集成起来。你们两个马上开始着手做这个事情。详细的功能列表我回头发个邮件给你们，一定把它们放进下一个冲刺啊。”

Timothy 说：“可以。冲刺进行到一半了。要移除哪些特性呢？”

Catherine 白了 Timothy 一眼。她早知道这么说没什么好下场。

刚才说话的时候，Dan 一直在盯着他的屏幕看。听了 Timothy 的话，他慢慢抬起头来。“你不觉得你们已经有足够的资源可以把新功能加入到当前的周期吗，Tim？”

Timothy 开始回答：“呃……我不这么认为，还有其他四个功能要做，而且其中一个的进度已经落后了……”

Dan 打断了他。“你这是借口。这些日子我根本见不到有谁加班或者真正拼命工作了。我今天早上 6 点就到了，公司除了我没有别人。你跟我说没法再多花点时间在工作上？这并不是什么大的功能需求。我自己一天就能编出来。”

Catherine 早料到会这样。这种对话她以前听过，当时也是不欢而散。他们勉勉强强把那些“好消息”请求做出来了，前提是略过了单元测试并且欠了一屁股技术债务。而用户注意到了，那一版发布后他们原本 4 到 5 星的评分掉到了 2 到 3 星。

“你就告诉大家加把劲，好好干。应该很快。我们会为此得到公司的赏识的。”

8.3 创造英雄与神奇思维

在经理中，有一派认为如果你给下属设定非常高的目标，这个目标将能够激励他们，而他们会更加卖力去实现该目标。如果你给每个团队成员都设置激进的目标和一个紧迫的期限，那么每个人都会挺身而出。他们似乎直觉上认为，这种“粗犷个人主义”的团队建设方法能消除官僚作风。每个人都被激发出解决其面临问题的能力，而这将使团队成为一个由能解决问题的人组成的高效团队。

粗犷个人主义的经理更赞赏英雄。那些每天工作到很晚，周末也加班，并且能够给出完整解决方案的人会得到最多的赏识，同时成为团队中职位最高的人。英雄开发人员之所以获得最高的职位并不是因为他们善于团队协作，也不是因为他们通过改进开发方式而让产品变得更好，而是因为他们用本该休息的时间完成更多的工作，而且是独立完成的。

这其实是不利于提高生产效率的。这的确是一种传统的思维，但历史反复表明，这种“个人主义”的思维将产出更糟糕的软件产品。为什么会这样呢？

团队协作才能创造出最好的软件。我们已经在 Scrum（自组织和集体承诺）和极限编程（创造一个让人精力充沛的工作环境，让大家形成一个整体团队）中看到了对团队协作的着重强调。很多现实生活中的敏捷团队反复地看到，改进团队协作和创造健康而放松的工作条件能够帮助团队开发出更好的软件。

既然如此，那为什么很多老板却喜欢招那些“粗犷的个人主义者”，而不去建设真正由具有团队精神和善于协作的人组成的团队呢？

我们不妨试着站在那些管理者的角度想一想。你很容易把开发团队看成是某种能够开发出软件的黑匣子。你告诉团队要做什么，过些日子，软件就做出来了。如果某个开发人员特别擅长独立工作，而且愿意在晚上和周末加班，他马上就会因为能够快速交付软件而引起你的注意。作为管理者，遇到这种人是你的福气：只管把工作交给他，他肯定给你干完。你肯定会奖励这个人，从而可能让其他人也像这个人一样努力工作。

好像你越是给团队施加压力，它就越能完成更多的工作。你越是奖励那些埋头苦干的人和在混乱中摸爬滚打的人，就越能开发出更多的软件功能。那么怎么给团队施加压力呢？你会让团队成员清楚总是有更多的工作要做，不管手头正在做的是什么工作，都要完成得越快越好。

我们应该体谅这个老板，即便他创造了一种缺乏活力的工作环境，让团队成员感到没有时间思考，只顾着做眼前的具体工作。我们通过学习极限编程有幸认识到了这种做法是如何导致团队编写出设计粗陋、难以维护的软件的。可是这位老板却并不这样认为。

这个老板所使用的就是*神奇思维*（magical thinking）。

在一个有着神奇思维的管理者眼中，一切皆有可能。开发团队可以应付任何项目，不管项目有多大。每个新项目都“不过是个小功能”或者“没什么大不了，我的团队能搞定”，因为一群受过良好训练的聪明人能做到任何事情。无论团队在这个星期完成了多少工作，

下个星期它都能完成更多。你可以不断给团队增加新任务，这些任务都会在不影响其他工作的情况下被完成。如果这意味着开发团队的成员那个星期需要加 20 个小时的班，或者有些人需要在周末加班，都无所谓，反正他们会把工作完成。就像魔术一样神奇。

神奇思维并非管理者的专利，它同样存在于英雄这边，他愿意加班创造“奇迹”，目的是得到认可、获得某个领导职位或者也许是得到更高的工资。他成为了评价其他团队成员的标杆。老板并不会去关注软件的设计和开发细节，也不会去注意有多少有设计缺陷的权宜之计变成了长久的解决方案。于是，每周工作多少个小时成了衡量一个团队成员价值的唯一标准。

神奇思维让人感觉良好。无论是“激励”团队的管理者还是那个努力工作并交付软件的英雄员工，你都会感觉自己解决了一个大问题。但是用神奇思维开发出来的软件从长远看是弊大于利的：技术债务积少成多，软件从来不曾“真正完成”，而且质量和测试总是被当作“锦上添花”的东西：bug 经常被引入，而且通常都是在软件发布后由用户发现。最终，团队的开发速度变得奇慢无比，因为团队花费在修复 bug 和糟糕代码上的时间远远大于它开发新功能的时间。

在 Scrum 和极限编程中，我们都提到过，能够快速开发出最好软件的团队，都有足够的时间去完成它的工作（通过限定时间的迭代），都能够每次只专注于一项任务，而且具备一个互帮互助共渡难关的协作环境。这就是为什么敏捷无法与神奇思维和英雄员工共存。

那么我们如何避免神奇思维呢？

这是精益的一个主要目标。不应该把团队视为“黑匣子”，精益思维帮助你理解团队每天的工作具体是什么。精益思维让你不只看到团队本身，它还让你清晰地看到项目从开始前到结束后的所有细节（神奇思维有时在项目结束后也会发生）。精益思维帮助你尽可能快地剥除那些善意的谎言，无论是老板对团队的、管理者之间的、还是我们自己对自己的那些妨碍我们构建最好软件的谎言。精益试图去除那些虚妄，并帮助团队共同协作，思考如何创造真正的价值，而不是单纯地蛮干。

8.4 消除浪费

有时候浪费现象并不容易发现。可能你不太容易注意到你和你的团队花费了大量时间去做的事情可能对项目来说没有什么实际价值。*消除浪费*（eliminate waste）这一精益价值观就是要找出项目中那些不创造价值的活动，并把它们剔除。

大多数团队并未真正思考过如何开发软件。一般团队成员间会有一种约定俗成的做事方式，新成员加入时会自然而然地接受。似乎没有人会真的静下心来审视我们是如何把软件从概念变成产品的（事实上，连思考一下团队是如何工作的可能都让人觉得怪怪的）。每个人都习惯于随波逐流。如果你一直都是在开始一个项目前先编写一个大型的规格文档，那么有一天让你试着不要搞这个文档，你会觉得不习惯。如果每个人都一直使用一个三年前开发的构建框架，那么你的下一个项目几乎肯定也会用同一个构建框架。

身处团队中的你，工作就是把软件交付出去，对吧？没有人真的有时间停下来去思考人生的意义，或者规格文档的意义。

在极限编程中，我们曾学习过，当团队习惯于不断对代码进行重构时，最终会得到更加灵活的设计。相同的思想也适用于思考整个团队是如何工作的这个问题：如果你习惯于不断“重构”运作项目的方法，你会得到一个更加灵活而能干的团队。

那么如何“重构”运作项目的方法呢？

重构代码的第一步是要寻找反模式。在精益思维中，项目运作的反模式被称为浪费。这很合理：浪费就是那些对构建更好的软件没有帮助的活动。

回头想想你最近做过的几个项目。你能想出你们团队做过的哪些事情对项目其实是没有帮助的吗？有哪些事情是每个人都被要求去做，但没有来得及去做，但依然感觉不应该不做的？你有没有写过从来没人看的规格文档？或者你是不是拿到了一个规格文档但从来没有看过它？你是否打算过要写单元测试和做代码审查，但不知怎么回事一直都没有做过？可能你做过代码审查，但是这些审查都是在临近软件发布的时候，所以大家都不敢提出问题，因为大家害怕会导致产品发布的延期，而审查过程中发现的问题不过是被记录下来以待日后处理。

通常一个项目中的很多工作，其实是不会让软件变得更好的。比如，一个项目经理可能在一个甘特图或什么其他项目计划上花费了大量时间，但该计划从未能准确地描述现实，因为它是基于估计和早期的信息，而从计划制订完成到团队开始开发工作这期间，这些估计和信息发生了重大的变化。更糟的是，这个项目经理可能事后又要花费大量精力去更新计划，让它与周期性的状态报告会保持一致。显然，这对软件本身没什么帮助，因为计划更新时，软件已经发布出去了。当然，可能这个项目经理并未打算把精力花在团队用不到的东西上。原因也许是某高级经理对任何延迟或超支都不能容忍。而其他人（包括那个项目经理）也许完全清楚这种做法对项目没有任何帮助。

这就是浪费的一个例子：项目经理在一个计划上花了精力，但该计划从未准确地反映现实，而且也并未被开发团队使用。并不是每个项目计划都是这样的，即便在瀑布式项目中。很多瀑布式项目有着靠谱的计划（或者，在不靠谱的组织里做到尽可能的靠谱）。但是，如果你以项目经理或开发人员的身份在这种项目中工作过，你就会发现这一反模式。这显然是一种浪费。

如果你冷静客观地审视一下你和你的团队每天在做的事情，就会发现各种各样的浪费。也许有一捆规格文档从未被打开过，在柜子上已经积了一层灰尘了。花在编写该文档上的工夫就浪费了。如果你一个星期花好几个小时做代码审查，排查出来的却不过是鸡毛蒜皮或个人偏好方面的问题，对设计毫无影响，也没有发现真正的 bug，那么这就是一种浪费。有些浪费甚至发生在你的项目开始之前，比如编写一个工作说明，让开发团队花一天时间去审阅，但是项目刚开始就把它扔到一边去了。你是否花了好几个小时去排查可以由脚本轻松解决的部署问题？这也是浪费。还有无用的进度报告会，每个人轮流报告自己的状态，以便项目协调人可以将其写进没人看的会议记录。浪费！

一个用每日站立会议替代了进度报告会的 Scrum 团队（即便只是得到“聊胜于无”的结果）就消除了这种浪费。从本书中你也已经看到上述其他浪费的例子可以怎样被消除掉。

某件事从软件开发的角度看是一种浪费，并不见得说它就一定没有用处。它只是对开发软件没有用处。也许某个高级经理需要那份项目计划，以便她可以说服股东继续为项目提供资金支持。或者政府监管部门要求必须做会议记录。工作说明可能对开发团队没什么用

处，但它可能是签署合同过程中的一个必要步骤。这些东西可能依然是必需的，但是它们对项目本身没有用处。这就使它们成为了浪费。

消除浪费这一精益价值观始于*发现浪费*，这也是该价值观的第一个思维工具。

通常我们很难把那些无用的活动视为浪费，因为它们几乎总是另外一个人需要重点考虑的事情：这个人可能是不属于团队一部分的项目经理，一个外包人员，一个高级经理等。有些浪费是大家都清楚的，可能每个人都知道做预算的过程很耗时，而且也对软件开发没什么帮助。其他的浪费由于太过熟悉而容易被人们忽略，比如你的团队分散在三个互不相连的区域，所以你得额外走五分钟才能与你的同事进行一次讨论。

Poppendieck 夫妇提出了“软件开发的七种浪费”这一概念。正如精益本身一样，这七种浪费也是从 20 世纪中期丰田公司的提法中借鉴来的。他们发现这些模式可以帮助你发现软件项目中的浪费。

- *做了一半的工作*
  当你在迭代的时候，你只交付那些“真正完成了的”工作，因为如果它没有 100% 完成，那么你就没有给用户创造价值。任何不创造价值的活动都是浪费。

- *多余的流程*
  多余流程的一个例子是第 7 章中提到的项目管理反模式：团队花费 20% 的时间汇报进度，做评估，所有这些除了用来更新一个进度表之外没有任何用处。所有花在这些跟踪和报告流程上的时间都不创造什么价值。

- *多余的功能*
  当团队开发了一个没有人要求过的功能而不是一个用户真正需要的功能时，这就是一种浪费。有时候发生这种事情的原因是团队中某个人对一种新技术十分热衷，想要借机会学习该项技术。这对于他个人来说可能是有价值的，因为他提升了自己的技能，这未来也许对团队也有一定的价值。但是这并不能直接帮助团队开发有价值的软件，所以这应该算作是浪费。

- *任务切换*
  有些团队被要求多任务并行，而这种多任务并行最终都变得不可收拾。团队成员感觉他们已经有一份全职工作（如开发软件），还被强加了很多额外的业余任务（如客户支持、培训等），而且任何一个任务都很关键、很重要。根据 Scrum 的专注这一价值观，我们知道，在项目间切换，或者哪怕是在同一项目的不相关任务间切换，都是会增加额外的延迟和精力的，因为切换情境会耗费人的脑力。现在我们有了另外一个词来描述任务切换：浪费。

- *等待*
  对于一个职业软件开发人员来讲，需要等待的东西太多了：等别人审阅规格文档；等着别人分配项目需要用到的系统访问权限；等待修复电脑问题；等待获取软件授权文件；等等。这些都是浪费。

- *移动*
  如果团队成员不坐在一起，那么需要讨论时，大家就真得站起来，走到对方那里才能讨

论。如果把这种不必要的移动加起来，可能会达到好几天甚至好几个星期。

- *缺陷*

  测试驱动开发可以避免很多的缺陷。每个热衷于测试的开发人员都曾经历过那个"惊喜时刻"：一个单元测试捕获了一个 bug，该 bug 如果没有被发现，后面会非常棘手，这时该开发人员意识到编写那些测试所花费的时间比定位该 bug 的时间要少得多，尤其是 bug 是在软件发布后由用户发现的情况下。另一方面，如果团队需要加晚班修复那些本可以被避免的 bug，这就是浪费。

我们称为"浪费"的这些东西里面是不是有一些看似是有用的？即使某件事情是一种浪费，它还是可以对某个人有用的（或者至少看起来是这样）。即使像开发人员座位分散这种问题，很可能方便负责办公室事务的行政人员。所谓的发现缺陷是要你理解这些做法背后的动机，并且客观地评估其重要性是否超过让项目高效完成的重要性。

不过，即使上述做法有它们的用处，但它们对于"开发一个为用户和公司创造价值的产品"这一目标来说就是浪费的。精益思维要求我们发现那些不为项目的具体目标创造价值的活动，无论是团队内部的还是外部的。

第 7 章中的框架陷阱就是团队自身难以发现的浪费的一个很好例子。一个原本可以用很少量的代码就解决的问题，开发人员却开发了一个大型的框架来解决它，那么这个框架就是一种浪费，这很讽刺，因为编写框架的原本目的是要通过把重复的任务自动化或者消除重复代码从而减少浪费。更糟的是，编写出来的框架常常会在将来变成一个障碍，因为只要团队需要该框架不支持的某个功能时，它要么需要扩展它，要么需要绕过它。为此而付出的额外努力也是浪费。

一个善于发现浪费的团队能够清楚地看到框架妨碍了为项目创造价值。在这样的团队中，团队成员能够理解浪费会影响到他们每天如何工作，并且能够清楚地看到日常工作中的浪费，即使这种浪费被认为对公司来讲是有必要的，他们还是把它当作一种浪费，因为它不会给项目创造价值。

用价值流示意图发现浪费

在《敏捷软件开发工具》一书中，Mary Poppendieck 和 Tom Poppendieck 推荐了一种简单的、用纸和笔即可操作的方法来帮助你找出浪费。这种方法叫作*价值流示意图*（value stream map），你可以给任意一个流程创建一个这样的示意图。跟其他精益技巧一样，价值流示意图源于制造业，但是它同样适用于软件团队。

为你的项目创建一个价值流示意图应该不会花费你超过半小时的时间。下面是具体的做法。首先从团队已经开发并交付了的一个较小的价值单元开始。试着找到一个尽可能小的单元，这是*最小可销售特性*（minimal marketable feature，MMF）的一个例子，也就是整个产品中，客户愿意给予优先级的最小功能单元。确定了这个单元之后，回头想想这个功能单元从设想到交付经历了哪些步骤。在纸上为每一个步骤画一个方框，用箭头把各个方框连起来。因为你画的是一个真正的功能特性在你的项目中所走过的路径，所以这将是一条直线，整个路径上没有决策和分支点，因为它代表的是该特性的真实历史。

MMF 这个概念在这里很重要。幸运的是，这个概念你已经学习过了。当一个 Scrum 团队的产品所有者管理积压工作表里面的条目时，那些条目一般来讲都是 MMF。一个 MMF 的形式通常是用户故事、需求或者特性请求。

接下来，估计一下完成图中的第一步需要多少时间；第一步完成后要等多长时间才能开始第二步。对每个步骤进行相同的估计，并且在方框的下方划线来表示工作和等待的时间。

图 8-2 是一个传统瀑布式流程中某个真实特性的价值流示意图的例子。

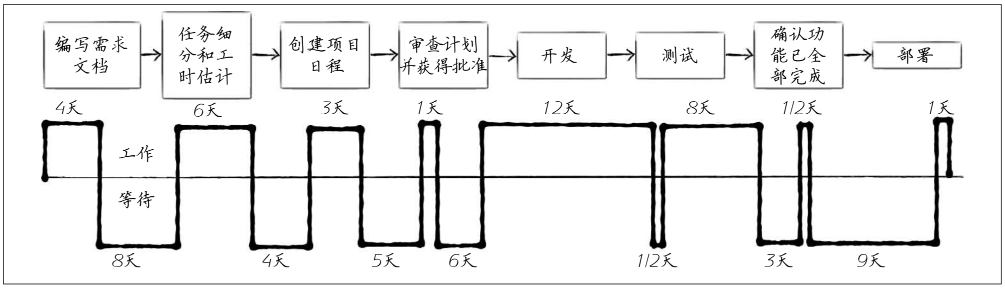


图 8-2：这张价值流示意图中显示的是一个软件项目如何在传统的项目管理周期中向前推进的。一个团队可以通过它来更清楚地看到时间都浪费在哪里了

价值流示意图清晰地显示了该流程中涉及多少等待的时间。从团队开始改项目到最终交付，一共花了 71 天。在这 71 天中，有 35.5 天花在了等待而不是工作上。这些等待时间可能是多种原因导致的：需求文档可能需要很长时间才能送交所有的评审者；或者工时估计会议必须延后，因为大家已经有其他安排了；或者计划审查是由一个委员会负责的，而该委员会一周仅召开一次会议。价值流示意图展示了对该特性的各种延迟所造成的累加效果，可是你不必被为什么会发生延迟的细节所困扰。看到这种整体的影响，可以帮助你进一步探究哪些延迟是浪费，哪些是必要的。

对于开发团队来说，也许看起来它并没花太多的时间等待计划被批准，因为它很可能在等待的过程中正在开发另外一个功能。但是需要使用该特性的客户并不一定清楚开发团队的优先级安排。而且，说实话，他也根本不关心，他只知道，他要求的功能从开发到交付一共经过了 71 天。

想象一下老板看到有这么多时间花在了等待上他会是什么反应，尤其是如果他不得不应付日渐急躁而愤怒的客户的情况下。如果团队能够找到一个方法减少浪费并缩短等待的时间，就可以大幅改善未来功能的交付时间。这会让客户更加满意，也会让老板更开心。因此，把浪费变得可视化能够更容易让每个人（尤其是老板）认识到有些改变是必要的。

8.5 加深对产品的理解

> 可感知的完整性指的是整个产品在功能、可用性、可靠性和性价比上达到的一种平衡，让用户感到满意。概念上的完整性指的是系统的核心概念相互协调，构成一个流畅、一致的整体。
>
> ——Mary Poppendieck，Tom Poppendieck，《敏捷软件开发工具》

具备精益思维，并不仅仅意味着能够清晰地思考如何工作和发现浪费，它还意味着对正在开发的软件产品有清楚的理解，并认识到产品是如何为用户创造价值的。当团队思考产品如何创造价值的时候，它所思考的就是完整性。这就要说到下一个精益价值观：*保证产品的完整性*（build integrity in）。具备精益思维的团队总是会考虑如何把完整性植入其软件中。

完整性实际上有两个方面：内部的和外部的。从用户的角度看，软件需要具备完整性（这属于外部的），同时，从开发人员的角度看，它也要具备完整性（这属于内部的）。精益包含两个思维工具来帮助我们理解内部完整性：*重构*和*测试*。它们与你在第 7 章中学过的完全是一回事。事实上，对于构建一个具备高度内部完整性的系统来说，一个非常有效的方法就是使用极限编程的实践，尤其是使用测试驱动开发、重构和增量式设计。

在本章中，我们将集中讨论外部完整性，也就是让软件对用户更有价值的东西。这需要我们理解用户是怎么想问题的。

这说起来可能有点抽象。不过幸运的是，精益有两个思维工具可以帮助你理解外部完整性。第一个工具是*感知完整性*（perceived integrity），即产品多大程度上满足了用户的需求以及该用户是否能马上感觉到他的需求得到了满足。

每一个优秀的产品都是用来解决一个问题或者满足一个需求的。有时候这个需求是很实际的现实业务，比如，某会计公司需要一个包含了今年税制改革内容的税务会计系统，这样他们的客户的税务减免才能合法；另外一些时候，需求会比较难以把握，比如，开发一个好玩的游戏。

如果一个软件 bug 多多而且经常崩溃，那它显然在感知完整性上存在问题。不过，一旦你通过了“软件能够正常运行”这条及格线，感知完整性就变得微妙起来了。举个例子，有一个大型新闻网站，多年来一直反复出现可感知的完整性方面的问题。有很长一段时间，把文字从文章中复制到文档或邮件中非常困难。一开始，选择文字进行复制粘贴时，会导致网站弹出一个窗口，显示的是选中文字的释义。最后这个自动释义功能被去掉了，从而允许用户复制粘贴。但是有一次网站改版，导致文本选择又不能用了，用户视图选择文本时会在一个新窗口弹出一篇相关文章。

这个网站的单击、选择、复制和粘贴等行为不合逻辑，让人抓狂。这些行为也许是有其目的的：新闻机构通常想阻止别人复制其内容然后粘贴到邮件中，更希望用户使用“用邮件发送本文”这类功能来分享文章。但是，有一个事实无法改变，那就是网站没能如用户预期的那样工作。这是感知完整性的一个反面例子。

第二个帮助你理解完整性的思维工具是*概念完整性*（conceptual integrity），即软件的各项功能在多大程度上形成一个统一的整体。一旦你理解了软件的感知和概念完整性，就可以让它对你的用户更有价值。

关于概念完整性如何影响了整个行业，有一个著名的例子，即 21 世纪头十年中视频游戏的演化。20 世纪末，大部分玩视频游戏的人都很在行。当时的休闲游戏玩家比现在少得多，他们中的很多人都很受挫，因为买回来最新的游戏后发现对他们来说太难了。而另一方面，铁杆玩家则经常抱怨很多新游戏太简单了。

接下来的十年间，游戏变得日渐流行。随着流行趋势的蔓延，游戏产业找到了同时满足两种用户群的需求的方法。这是如何做到的呢？

开发人员首先意识到普通玩家和铁杆玩家都需要游戏具备概念完整性。像《俄罗斯方块》《愤怒的小鸟》和《糖果传奇》这种好玩、休闲的游戏需要一关一关地慢慢增加难度。休闲玩家很重视这种稳定的难度提升，同时持续地感受到一种成就感。如果《愤怒的小鸟》开始的 5 个关卡非常简单，然后就是一个特别难的关卡，用户就会不玩了，因为这显然有悖于概念完整性。我们把这种对概念完整性的违背称为“不一致”。

铁杆玩家不喜欢这种节奏缓慢稳定的游戏，也不稀罕那种不是他们努力得来的晋级。他们常常更能够从“折磨”中获取满足感，在所谓的折磨中，他们必需要完成重复性的，而且经常是折磨人的任务，才能得到晋级。

一个娱乐性的游戏不应该有折磨人的关卡，而一个折磨人的游戏则不应该有简单的关卡。像 *Flappy Bird* 和《超级肉肉哥》这类游戏以及很多终极幻想系列的游戏，就因为它们的难度而被铁杆玩家所追捧，这些游戏的一个关卡常常需要很多次的反复尝试才能通过。折磨人的游戏出现简单关卡可能会带来与简单游戏中的困难关卡同等的不一致。

在 21 世纪头几年游戏产业的发展过程中，游戏开发团队在概念完整性方面遇到了很多问题。很多游戏收到了很多负面评价，要么因为它们对于铁杆玩家来说太简单了，要么因为它们对休闲玩家来说太难了。在这个过程中，这些游戏开发团队学会了在游戏中添加一些特性，从而提高游戏对两种用户群的概念一致性。例如，大部分同时针对两个用户群的游戏都会有一个难度设置。如果你在游戏中的角色不断挂掉，系统就会提示你调低难度。铁杆玩家绝不会选择这个选项，具备良好概念完整性的游戏也不会不停地问他是否要调低难度，因为问这个问题本身就与一个难度高的游戏不一致。不过，游戏业界也有一种观点认为休闲玩家和铁杆玩家代表着两个完全不同的市场，很多游戏仅针对其中一个进行营销。

所有这些进展都是关于提升游戏带给玩家的价值的，而这种提升是通过理解玩家是如何玩游戏以及保证难度等级的概念完整性来做到的。游戏开发团队改变了其工作方式，改变了其设计游戏的方式，以此来提升概念完整性。现在，软件团队普遍在项目的一开始就决定其受众是休闲玩家、铁杆玩家，还是两者兼有。它会针对目标受众安排测试任务，并且与市场部门配合以确保游戏的宣传针对的是正确的受众。这些都是一个团队如何改变工作方式以便提升概念完整性的例子。

8.5.1　着眼全局

在一个软件团队中工作，你绝不是身处真空之中。公司和团队的组织方式会对你的工作方式产生很大的影响。而且，在任何一个组织中，都会存在一些可能影响到你的项目的障碍和阻力。比如，在开始做一个新功能之前，你可能需要经理批准一大堆规格文档；也可能用户的几条负面评论就导致产品所有者惊慌失措，开始给你在周末安排工作；你的老板可能迷恋任务管理系统中过度复杂的工作流，所以你得把一个票证的状态修改 8 次才能开始处理它。这只不过是几个例子，也许你能够想到一些你自己的关于这类深植于你工作方式中的低效率活动的例子。

我们前文已经提过这些东西都属于浪费，不过我们也谈到了，有时候这些做法是有它的目

的的，也许不会直接让你的项目收益，但是公司和项目需要它们。例如，团队可能会浪费时间（单纯从项目的角度看）填写对项目没什么帮助的报告，但是如果这些报告是监管者的要求，那么它们对公司来讲就是值得的。那么，我们怎么能够知道哪些活动是确实有用的呢？

这就要提到下一个精益价值观：*着眼全局*（see the whole）。要真正了解你的团队是否在高效率地工作，你需要退后一步并理解整个系统，因为有时候人很容易对某个方案存在情感上的投入，所以你需要对全局进行一个客观的审视。比如说有一个项目经理搞了一个时间表，要求每个开发人员每天填写一个以 15 分钟为单位的时间表。有了这个时间表，她就可以随时了解团队的状态，她可能对此感到很满意。但是她可能并没有意识到这给团队增加了多么大的负担。如果你能证明这种做法会降低团队 5% 的生产效率，那么就更容易说服该项目经理取消这种不必要的做法。

不过需要注意的是，所谓的“理解团队所处的系统的特点”可能听起来很简单，但做起来可并不容易。团队中的每个人对项目的看法是好还是坏有很多影响因素，这可能取决于做这个项目是否有成就感，或者取决于他们如何看待自己对项目的贡献。比如，如果一个开发人员解决了一个很有意思的编码问题，她可能会感到做这个项目非常有成就感；再比如，如果团队总是能够在截止时间之前完成任务，那么项目经理肯定会对该项目更满意，但是如果开发人员为此需要每天晚上和周末都加班，那她可能就会觉得项目不够成功。

这就是为什么精益团队通过*量化指标*（measurement，这是另一个精益思维工具）来统一大家对项目的认识。可以用来作为指标的东西很多，而选取正确的指标能够帮助团队更好地了解项目。根据要解决的问题的不同，精益团队会选取不同的指标。项目中的每个人看问题的角度都有所不同，而客观的量化指标可以让大家看待项目的时候有一个共同的参照系。

对于没有使用量化指标衡量一个系统（尤其是软件团队进行软件开发的系统）的人来说，这听起来可能很抽象。我们不妨用一个具体的例子来说明一下。

用户非常在意软件团队对其需求的反应是否迅速。商务人士和用户希望他们提出的功能请求能够快速地添加到软件中。与为三个功能等待三个月相比，他们更愿意每个月得到一个新功能。这就是 Scrum 和极限编程都用到迭代方法的原因，也是敏捷团队缩短反馈循环的原因。

假设你是一家公司的老板。你的公司有一个软件项目。你怎么知道自己的钱是否花在了刀刃上呢？假如说你的项目经理定期提交给你有关项目进度的报告，告诉你各项任务都在按时推进，开销也都在预算范围内，一切状况都很好。他提供了各种汇总、报告以及满是绿色圆点的项目计划，上面写满了已完成的任务，所有这些材料都说明最近的四次发布几乎是 100% 顺利的。

这听起来似乎是不容置疑的证据，不仅证明项目团队的工作得力，而且也证明它在交付用户请求的功能。更进一步地，日程表中留出了足够的余地，有清晰的风险登记表，还有一个任务跟踪系统，这就说明项目完全在控制之内，无论是将来可能出现的已知或未知的风险，当前计划都已经纳入了考量。作为老板，这会给你一种温暖模糊的感觉：项目在你的掌握之中；你很了解项目的运作方式；而且你能够处理预期之外的问题。

可是，如果你了解到你的几个客户好几个月之前提出的简单要求，到现在还没有加到软件中，你会怎么想？如果你的客户开始换用竞争对手的软件，因为竞争对手能够更快速地响应用户的需求，你会怎么办？你还会认为你的项目很成功吗？显然，项目存在严重的问题，而你需要与开发团队一起来解决它。

你怎样让开发团队对重要的请求作出最快的反应呢？假如说你尝试跟团队对质，可是团队成员可以拿着进度报告，告诉你项目进展一切顺利。你如何让他们意识到问题的存在呢？

量化指标可以帮上忙。很多团队使用*交付时间*这个指标。这个指标是指一个特性从被提出到交付所用的时间。

具体的算法是这样的。每当用户提出一个特性请求，记录下这个开始日期。当包含这一特性的软件版本发布时，记录下这个结束日期。开始日期和结束日期之间的时间差就是该特性的交付时间。把某个发布版本所有特性请求的交付时间求一个平均值，就是发布版本的平均交付时间。[3]

如果你问一问团队成员，他们认为的平均交付时间是多少，你猜他们会怎么说？如果团队每个月发布一个版本，他们很可能会猜交付时间在一个月到两个月之间。这个交付时间也许是可以接受的，你的大部分用户可能对此感到满意，即使有抱怨，也是来自极少的一部分用户。

可是，如果交付时间算出来发现远远超出你的用户所能接受的范围呢？比方说连一个非常简单的用户请求都要经过六个月才最终交付，这显然是不可接受的。是开发团队的错吗？还是你这边的某些做法有问题？或者说在当前的工作方式下，这么长的交付时间是无法避免的？你现在还无法回答这个问题。但是你清楚有问题存在，而且由于有量化指标可供参考，你可以帮助团队认识到问题的存在。

现在，如果你跟团队坐下来谈用户的抱怨，如果项目经理指着进度报告告诉你不存在问题的话，你就可以拿出交付时间这一客观指标，证明问题是存在的。这比粗暴地说“我是老板，听我的”要好得多，因为现在大家都有一个可以为之努力的清晰、客观的目标。这不是随意的决定，也不是神奇思维。

8.5.2　找到问题的根本原因

通过量化指标来对项目和团队进行客观的观察只是着眼全局的第一个部分。第二个部分则涉及理解问题的*根源*，即导致问题发生的真正原因。

在第 6 章的结尾，我们提到过我们会在后面再次回到根本原因分析这个话题上，因为根本原因分析不仅是精益思维的重要部分，它也是极限编程团队的衍生实践之一。极限编程团队和精益团队都通过自问*五个为什么*找出问题的根源。与精益思维的其他部分一样，这个技巧也源自日本汽车制造业，不过它在敏捷团队扎下了根。这个技巧很简单，首先问一下为什么问题会发生，回答了第一个为什么之后，接着问下一个为什么（一般总共问五次），

注 3：交付时间还有其他计算方法，比如，你可以通过加权给大功能更多的权重。我们这里只是要说明量化指标的用处，所以选择了一种简单的计算方法。

直到找出根本原因为止。

在我们上面的例子中，那个开发团队就可以使用“五个为什么”技巧来找出交付时间过长的原因，具体来说，团队成员可以问以下这些问题。

- *为什么平均交付时间这么长？*因为大多数用户提出的功能需求从提出到完成需要半年的时间。
- *为什么用户的需求要经过半年时间才能加入到软件中？*因为这些需求总是被推迟，以便给临近发布时的修改腾出时间。
- *为什么临近发布了要做那么多修改？*因为在发布软件之前，开发团队需要与高级经理一起进行一次审核，而经理几乎总是会要求进行一些涉及根本的改动。
- *为什么经理总是要求进行这种涉及根本的修改呢？*因为他们对软件的外观、功能甚至所需使用的技术工具都有非常具体的看法，但是开发团队在已经完成开发并给这些经理做演示的时候才得知他们的这些看法。
- *为什么开发团队在软件开发完成后才得知这些看法呢？*因为高级经理太忙了，在项目早期没有时间跟开发团队交流，所以他们只参加最终的演示，于是软件开发完毕后开发团队还得再重新考虑很多基本问题。

噢！现在我们知道为什么团队需要花费那么长的时间来响应用户的请求了。通过使用量化指标和寻找根本原因，我们发现这完全不是开发团队的错。原来开发团队完成了很多特性的开发，但是当向高级经理演示的时候，高级经理会要求开发团队做很多的修改。也许这些修改是必要的，经理的想法也确实是好的。可是就算是必要的修改，也需要项目经理进行影响分析、更新项目计划，同时把受到影响的特性安排到后续的某个发行版本中。这就是导致交付时间过长的原因。还有更糟的，有些特性请求已经安排到下一个版本中了，现在它们又得被推迟到再下一个版本中，这样才能给新增的修改任务腾出时间。所有这些造成的结果就是，部分用户换用了竞争对手的产品。

理解了交付时间过长这一问题的根本原因之后，我们就可以想出一些解决问题的方案。比如，开发团队可以进行迭代式开发，让经理参加每次迭代结束时的演示，而不是仅仅参加大版本发布前的演示。或者，经理可以把他们的批准权委托给一个能更频繁地参与项目并与团队沟通的人（像 Scrum 中的产品所有者），并且相信这个人会作出正确的决定。再或者，开发团队和经理可以保持现在的开发方式不变，但是增设一些账户经理，让他们与用户和客户做好沟通和解释工作。

总结一下：开发团队从一个问题开始，这个问题是对用户需求响应不够及时。通过使用量化指标和寻找根本原因，它做到了着眼全局。开发团队理解了项目在整个公司中的位置，并且能够找出几种方案，来长远地解决交付时间过长的问题。最重要的是，开发团队与老板现在都能够得到客观的信息，并且可以一起作出决策。

8.6 尽快交付

还有一则精益价值观，就是*尽快交付*。

看到“尽快交付”的字眼时，你首先想到的是什么？你是否想到一个周扒皮似的老板或项

目经理，一个劲儿地催促团队加晚班，快点把软件交付出去？你是否以为“尽快交付”就意味着要把测试、次要功能以及所有可以看作是“多余的”或次要的东西全都抛弃？也许这个词会让你联想起一个英雄开发人员晚上和周末不停加班，为了尽快交付一个重要功能而编写快速但粗糙的代码。大部分的经理听到“尽快交付”这句话的时候，都会冒出类似上面的想法。甚至很多开发人员、测试人员以及其他软件工程师也会有相同的想法。

敏捷团队知道上述的那些做法会导致你的团队交付速度变慢，而不是变快。所以我们提倡可持续开发。（“赞助商、开发人员和用户要能够共同、长期维持其步调，稳定向前。”，这是我们在第 3 章学过的原则之一。）偷工减料和延长工作时间这些做法只会得不偿失。让团队有足够的时间去做正确的事，从而可以更加快速地交付更好的工作成果。

尽管上面所说的都是事实，但似乎还是比较抽象。Scrum 的“专注”原则和极限编程的“精力充沛的工作”实践就把这一点具体化了。Scrum 和极限编程让我们看到如何通过迭代和流程来现实地达到这种最佳的交付节奏。精益在这个基础上进一步为我们提供了三个思维工具，来帮助团队做到尽快交付：拉动式系统，队列理论和延迟成本。

队列理论的目的是要保证人们不要超负荷工作，以便保证他们有足够的时间按正确的方法做事。所谓队列，就是包含团队或单个开发人员的所有任务、特性或者待处理事项的列表。队列里的东西是有顺序的，这个顺序通常是先进先出，意思是说，除非有人特别修改了顺序，否则在队列里面存在时间最久的那个条目就应该是下一个被拿出来处理的条目。队列理论是对队列的数学化研究。队列理论研究的一个领域涉及预测早期添加的队列对系统的最终结果会有怎样的影响。精益告诉我们，让团队的任务队列公开化，并且把它作为决策过程的核心，可以帮助团队更快地交付软件。

要点回顾

- 相信团队能够做到不可能的事情，并且忽略现实世界的限制和项目的实际情况，这种思维就称为**神奇思维**。
- 精益团队**消除浪费**的方法是：找出那些对开发一个有价值的产品没有直接贡献的活动，并移除它们。
- 任何不直接为项目创造价值的活动都是**浪费**，精益团队致力于发现这些浪费，并尽可能将其从项目中消除掉。
- Mary Poppendieck 和 Tom Poppendieck 提出了**软件开发的七种浪费**：做了一半的工作、多余的过程、多余的功能、任务切换、等待、移动和缺陷。
- **保证产品的完整性**是另外一个精益价值观，它包含**感知完整性**（即产品多大程度上满足了用户的需求）和**概念完整性**（即产品的各项特性多大程度上能够有机地结合在一起）。
- 精益的思维方式帮助团队着眼全局，客观理解团队的工作方式，包括其中的问题；量化指标能够帮助你对你的项目和团队有一个客观的看法。
- **尽快交付**意味着去除那些延迟你的工作和导致瓶颈的无用活动。

8.6.1 使用面积图可视化工作进度

怎么才能知道你是否做到了尽快交付呢？

精益思维可以回答这个问题：使用量化指标。一个衡量你的团队是如何交付有价值产品的有效方法是使用*工作进度面积图*（work-in-progress area chart），简称为 WIP 面积图。这是一个简单的图示，它显示了最小可销售特性是如何在你的价值链条中向前推进的。

如果你创建过一个价值流示意图，那么你就可以创建一个工作进度面积图，后者用于表示特性、产品和其他价值指标是如何经过价值流的每一个环节的。这种方法最适合于使用 MMF，因为后者代表了最小的价值单位。

我们来看一个价值流示意图的例子，该图显示了一个网站开发公司如何处理大部分 MMF。

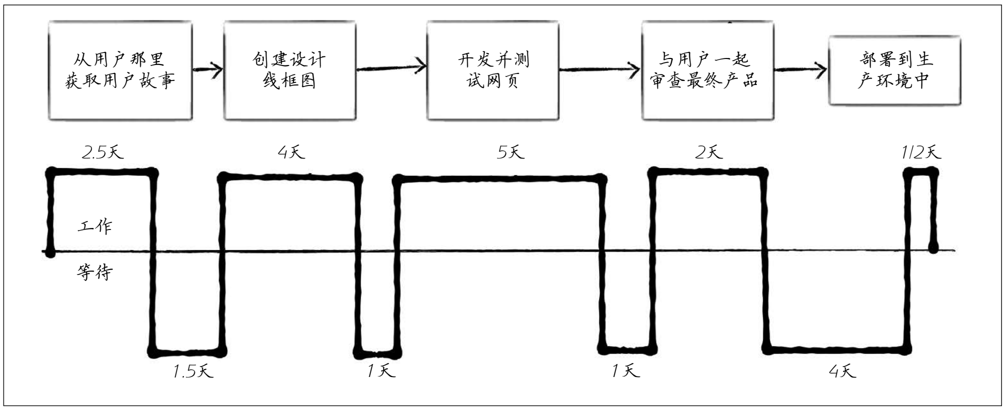


图 8-3：我们将使用这个价值流示意图来构建一个工作进度面积图

工作进度面积图的目的是要展示*进行中的工作*（work in progress，即团队正在开发的所有有价值的特性）的完整历史。这个图会告诉我们，在任何一个日期，有多少个 MMF 正在开发中，这些 MMF 分别分布在价值流的哪些阶段。这个工作进度是针对特性的，而不是针对任务的。换句话说，它显示的是有多少个产品特性正在开发中，而不是实现这些特性时所需完成的具体任务。在第 5 章中，你已经了解了特性与任务之间的关系（团队把用户故事分解成具体的任务，后者则需在任务板上挪来挪去）。用户故事是表示一个 MMF 的很好方法，因为它是可以交付给用户的一个小型的、自包含的价值单元。用户故事会出现在工作进度面积图中，但是与之相关的那些具体任务则不会。

工作进度面积图的具体画法是这样的：首先以日期作为横轴，以 MMF 的个数作为纵轴；价值流示意图中的每个方框在工作进度面积图中都有对应的一条线；这些线把整个图分成了多个区域，后者则与价值流图中的方框对应。

项目一开始的时候，正在处理中的 MMF 的个数是零，所以在 (0, 0) 处有一个点。假如说项目开始时，团队同时开始处理 9 个用户故事，并将用户故事用作该项目的 MMF。几天后，团队又增加了三个用户故事。那么你应该在 (1, 9) 那里画一个点，然后在新增加三个 MMF 那一天的 12=9+3 那里画一个点。然后，你需要把这些点连起来。

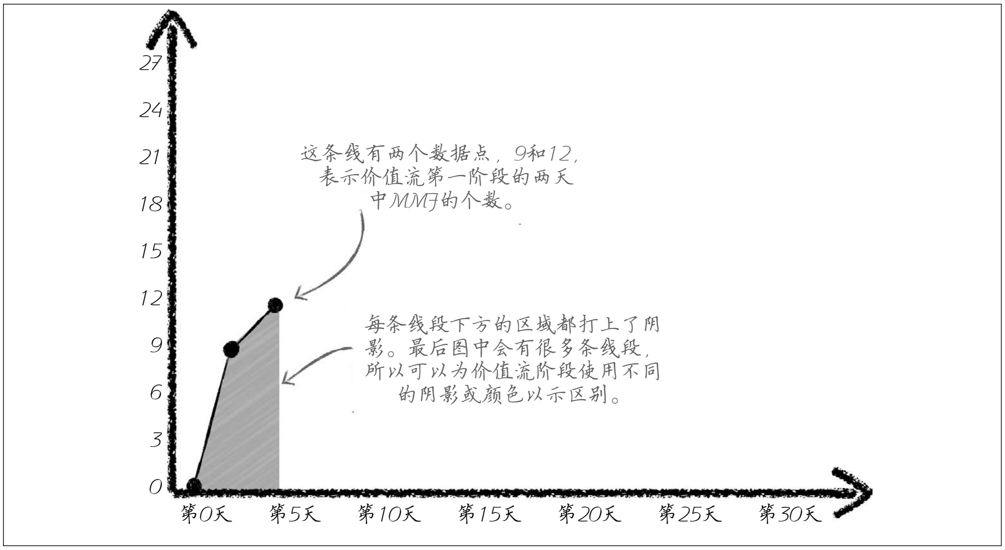


图 8-4：开始构建工作进度面积图：绘制价值流第一阶段的 MMF（像用户故事）的折线图，然后给下方区域加上阴影

几天后，程序员开始为 4 个用户故事创建设计线框图了，所以这四个用户故事进入了价值流的下一个阶段。系统中 MMF 的总数仍然是 12，但是现在它们被分成了两部分：价值流第一阶段有 8 个还在处理中或者等待中，因为价值流图同时记录了工作和等待时间，另外有 4 个进入了第二阶段。这样，当前日期处应该在 4 和 12 两处都画上一个点。

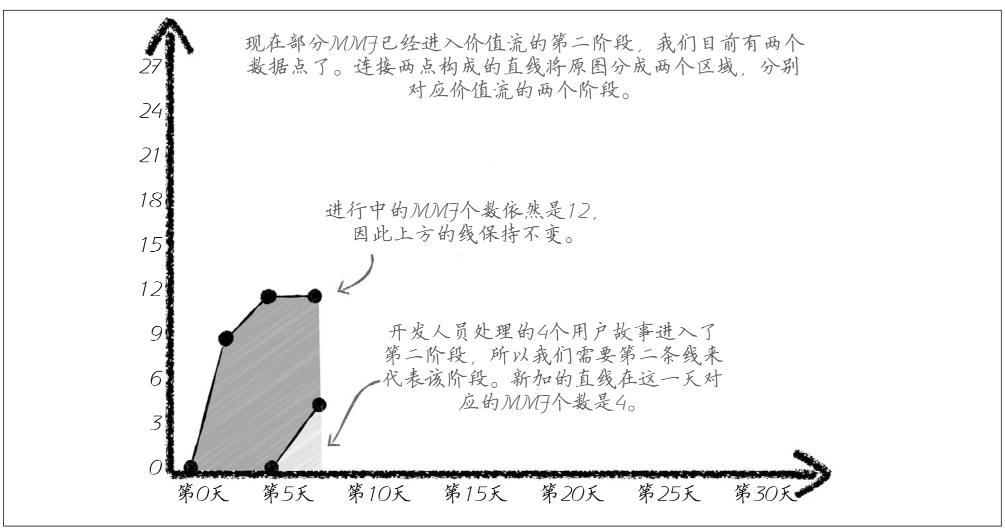


图 8-5：当工作进入到价值流的第二阶段时，工作进度面积图上需要添加与该阶段相对应的一条线，这样就把原来的面积图一分为二。上面的那条线依然代表进行中的全部 MMF 数量，它与新增的那条线之间的高度则表示处于第一阶段中的 MMF 数量

随着 MMF 在价值流中向前推进，任务总数不断增加，工作进度面积图中将逐渐增加与价值流图中每个阶段相对应的带状区域。

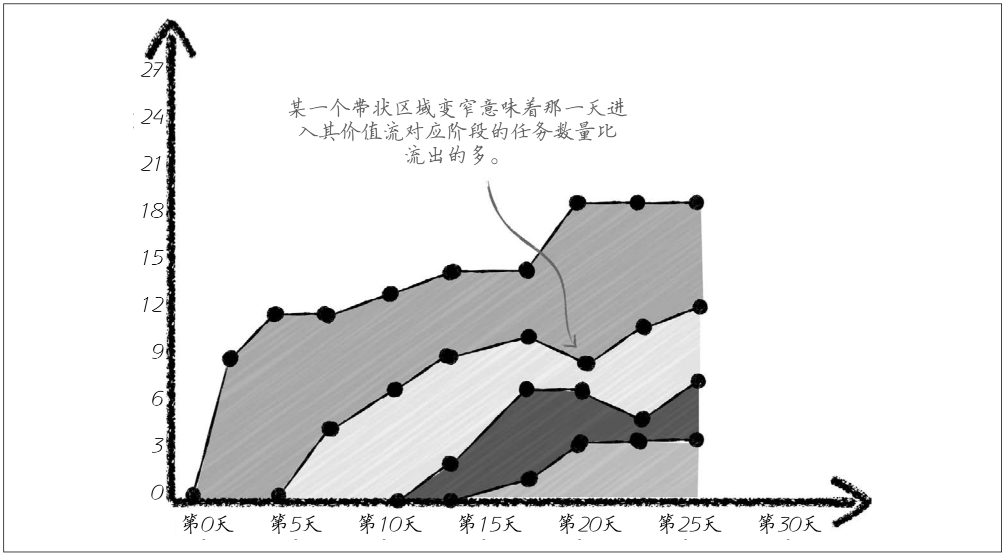


图 8-6：工作进度面积图显示了进行中的工作随时间的变化情况

MMF 完成后该怎么做呢？如果你把它们画在图中，那最终“已完成”的 MMF 的数量将增长到一个超大的值，以至于与其他值完全不成比例了。这会使代表进行中的 MMF 的那些区域小得好像山顶的彩带，几乎看不见了。

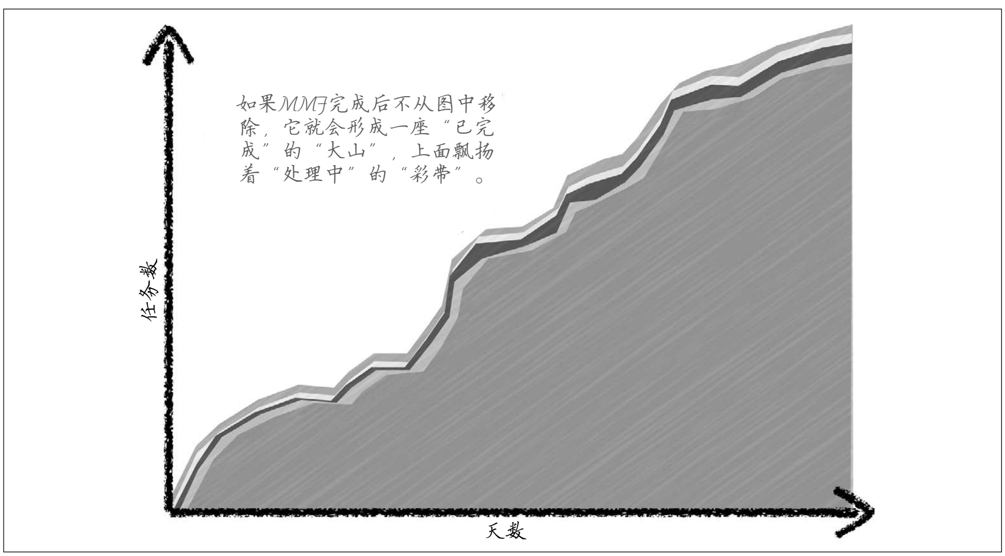


图 8-7：当团队想给老板看看完成了多少工作时，就把“已完成”的 MMF 画在图上。这看起来很让人印象深刻。糟糕的是，这么做会让该图难以量化进行中的工作

这样做是在强调增长，而不是强调流程。虽然这种能让上级感到印象深刻的图表是很好的进度报告，因为它显示出团队完成了大量的工作，但是它对于管理工作流程并不是特别有用。把“已完成”的工作从图中拿掉能够更清楚地呈现出价值是如何在整个链条中流动的。

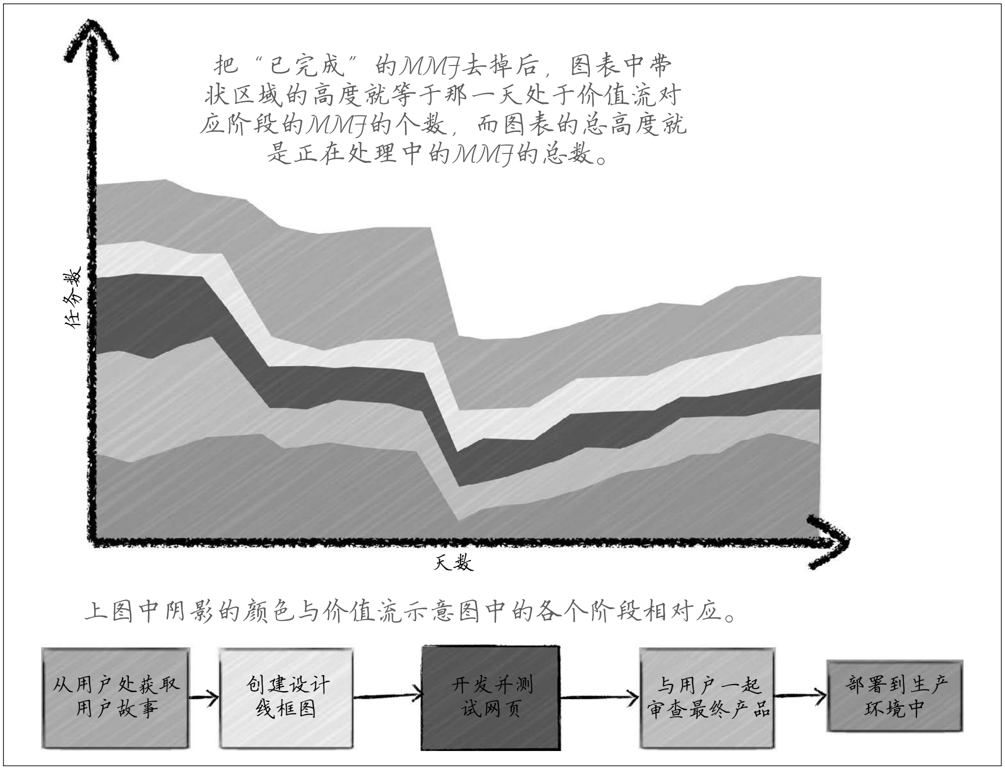


图 8-8：把“已完成”的功能从图中移除会比较好。另外可以用不同的颜色来绘制图表，以便更容易区分价值流的不同阶段

正因如此，大部分的工作进度面积图都不包含已完成的功能。这样一来，如果你的项目推进速度逐渐稳定下来，你的工作进度面积图也就稳定下来了[4]。当一个 MMF 从价值流的一个阶段进入下一个阶段时，图中上一阶段所对应的带状区域就会变窄，而该 MMF 进入的那个阶段所对应的带状区域会变宽。这样就很容易看出趋势，比如什么情况下大量的 MMF 从一个阶段进入下一阶段（或者移出整个价值链条）。

注 4：看到上面这个图你是否会想问我们为什么不把这个图叫作累积流图（Cumulative Flow Diagram，CFD）？在第 9 章中我们会解释 CFD 与工作进度面积图的区别。

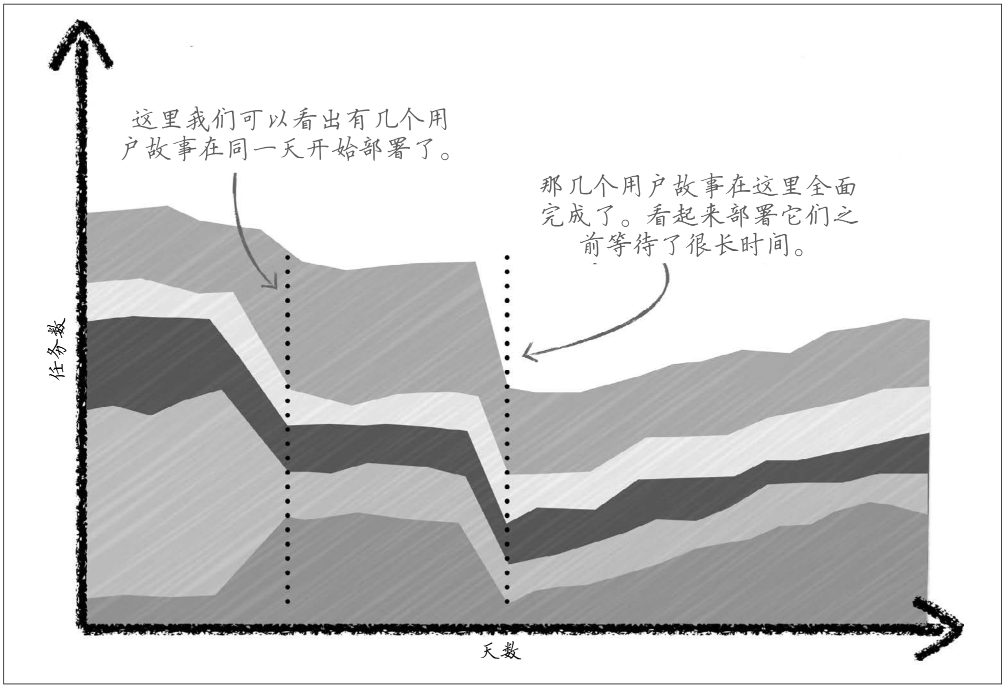


图 8-9：工作进度面积图帮助你看到各项工作是如何在价值流当中流动的，并且让你更容易发现延迟和其他可能的浪费，比如那些花了很长时间才部署到生产环境中的用户故事

8.6.2 限制进行中的工作，控制瓶颈

一个用到队列理论的重要思想称为“约束理论”。该理论的提出者 Eliyahu M. Goldratt 是从物理学家转行做管理的。限制理论的一个观点认为系统中的某个特定的约束（比如负担过重的团队积累起很多未完成的工作）会对系统能够完成的总工作量造成限制。当该约束被解除后，另外一个约束就成为决定性的了。约束理论告诉我们：每一个超负荷的工作流都至少有一个约束。

当某个约束导致工作流的某一个点产生了工作积压时，人们往往把它称为*瓶颈*。从系统中消除一个瓶颈（可能通过改变流程或增加人手的方法）能够使得工作的推进更顺畅。约束理论告诉我们系统别的地方还会有另外一个决定性的约束。但是，我们可以通过系统性地发现并消除关键约束来减少总的浪费。

在一个必须每天面对这些约束的团队中工作会有什么感觉呢？换句话说，当你和你的团队就是系统的瓶颈时会有什么感觉？

当你就是系统的瓶颈时，其他人就会希望你能够进行多任务处理，不断地在你正常的全职工作与很多其他小“兼职”之间切换。应当注意的是，很多健康的团队在同一时刻也有很多的任务要做，但是它们并不把这称为“多任务处理”；人们一般使用“多任务处理”来掩盖团队其实在超负荷运转这一事实。把工作细分，然后让团队进行多任务处理，这种做

法常常会让你意识不到核心问题其实是你的时间不够用，尤其是当你把任务切换的额外时间和精力考虑进去的时候。

例如，一个已经把 100% 的时间花在开发工作上的团队，可能被一个有着神奇思维的老板要求进行“多任务处理”，让团队成员每周再花好几个小时在客户支持、培训、维护、会议以及其他工作上。对于该团队来说，可能很难意识到它的时间不足以完成那么多工作，尤其是当那些额外的工作是一点点加上来的时。它只会开始感觉到自己超负荷了，可是因为这种工作方式称为“多任务处理”，它可能未必意识到它为什么会有超负荷的感觉；它可能只会感觉有太多“兼职”工作，有点顾不过来。现在我们知道，在团队工作流程的某个地方，越来越多的工作积压起来，而这些积压起来的工作给团队增加了沉重的负担。

8.6.3 拉动式系统帮助团队消除约束

> *我始终遵循一个宗旨。每个与我共过事的人都清楚；我把这个宗旨挂在墙上：“面对愚蠢的入侵，消灭它是你的道德责任，不管给它说好话的人是谁。”*
>
> ——Keoki Andrus，《团队之美》（第 6 章）

所谓*拉动式系统*，指的是通过使用队列或缓冲区来消除约束的一种运作项目的方法或流程。它也源自日本的汽车制造业，并且在软件开发中也得到了应用。汽车制造商（更具体地说是 20 世纪五六十年代的丰田）开始研究它们的配件库房，并且尝试寻找降低配件库存的方法。经过大量的实验，它们发现即使库房中有组装一台汽车的几乎所有零件，只要少了哪怕有限的那么几个零件，整个生产线就无法开动。整个装配团队就得等着，直到那些缺少的零件送达才能开工。*延迟成本*（cost of delay）就变得很重要：如果一个零件缺货，那么将该零件送达装配线的延迟就非常昂贵；如果该零件比较充裕，这种延迟的成本就低得多。

丰田的团队发现，如果它能够知道装配团队需要哪些零件，并且仅把这些零件送往装配线，它就能够削减成本，同时更加快速地交付成车。为了解决这一问题，团队想出了*丰田生产系统*（Toyota Production System，TPS），这一系统就是精益制造的前身，而 Poppendieck 夫妇采用精益制造创建了精益软件开发。

TPS 的核心思想是，工作流中有以下三种制造约束的浪费必须要消除掉。

- Muda，意思是徒劳，闲置，过剩，浪费，损耗，废物。
- Mura，意思是不平衡，不规则，缺乏一致性，不平均。
- Muri，意思是不合理，不可能，超出某人的能力，太难，强制，不得已，强行，强迫，过度，无节制。

任何参与过管理混乱、运营不善的软件项目的人（尤其是使用过低效工作流程的人）对徒劳无益、不均衡以及不合理的做法都非常熟悉。这对于低效率的瀑布式流程自然不言而喻，不过，对于那些曾经在一个勉强采用 Scrum 或极限编程实践而仅得到了“聊胜于无”结果的团队中待过的人来说，这些东西应该也不陌生。

你是否对下面这些感到熟悉？

- 让每个人确认某规格文档要花很长时间，而与此同时开发人员则坐在那干等着项目开工。甫一开工，他们就已经落后进度了。
- 管理层争取项目预算耗时太长。等到资金批下来，进度已经落后了。
- 开发到一半，开发团队意识到软件设计或架构的一个重要部分需要更改，但是这会导致非常严重的问题，因为有很多其他部分依赖它。
- 质量控制团队要等到每一个特性都开发完毕才开始测试软件。质量控制团队发现了一个严重的 bug 或者一个严重的性能问题，而开发团队不得不进行抢修。
- 分析和设计花费的时间过长，导致进入编码阶段时，每个人都需要加班加点地赶工期。
- 软件架构师设计了一个庞大、华丽、复杂但却没法实现的系统。
- 即使是对软件规格、文档或者计划的最小修改都需要经过一个冗长的修改控制流程。大家为了绕过该流程，于是甚至把大型的、颠覆式的修改都放到 bug 跟踪系统里面。
- 项目进度落后了，于是在最后几周时间里，老板给团队增派了人手。最终并未让项目进展快起来，这种做法反而导致了各种混乱。[5]

回想一下你自己的那些因为某些愚蠢的做法而出现问题的项目。这些愚蠢的做法可能是你不得不接受的愚蠢规矩，或者是老板强制的，或者是你公司文化的一部分；也可能是愚蠢的、不必要的、专门用来“激励”你的截止时间。

这些做法的出现并非偶然。花点时间回头看看 Muda、Mura 和 Muri 的定义，然后再看一遍常见的项目问题列表，还有你在自己的项目中曾经遇到过的问题。看看你能不能把这些问题分别归入 Muda、Mura 和 Muri 这三个类别中。你不得不做的事情中，有没有哪些是无用的或多余的？这类事情就属于 Muda，就是你不得不做，却不创造价值的事情。有没有一些时候，你只是闲坐着不能开工，焦急地等待某人给你答复？这就是 Mura，不均衡，工作总是一阵多一阵少。是不是还有些时候，你不得不加班加点，因为你被要求完成比你所能承受的大得多的工作量？这就是 Muri，超负荷，被要求做到不合理或不可能的事情。

尽管软件开发与汽车制造有着很多的不同，Poppendiecks 夫妇意识到了这些精益制造的思想也能够影响软件项目。因此，可以大胆地推理，如果软件团队面临的问题与制造业的问题类似，那么那些对制造厂商起过作用的解决方案很可能也会对软件团队起作用。对于制造业来说，这个解决方案就是拉动式系统（也叫“即时生产”）。

在 20 世纪 50 年代，由 Taiichi Ohno 领导的丰田制造团队就已经意识到，要提前预测哪些零件在将来会发生短缺是非常困难的，因为貌似发生短缺的常常是不同的零件。这是浪费（Muda、Mura 和 Muri）的一个重要的根源。于是，这个团队设计了一个系统，在该系统中，如果装配线上的某个岗位需要更多的零件，该岗位的工作人员就发出一个信号，有一个专门负责运送零件的团队，会根据这些信号来决定把哪些零件运送到哪些岗位。每个岗位都有一个队列，用来存放需要的零件。这样一来，就不是由库房向装配线“推送”零件，而是装配线从库房那里“拉取”零件，而且仅在零件面临短缺的时候才进行这种拉取。

这种系统被称为“拉动式系统”，因为它由相互独立的团队或团队成员组成，各个团队或

注 5：在《人月神话》中，Fred Brooks 给出了所谓的 Brooks 法则：“为延期的项目增派人手，只会让它进一步延期。”在你看到 Muda、Mura 和 Muri 的时候想想 Brooks 的这句话。

团队成员只拉取各自需要的那些零件（而不是任凭一大堆零件推送给他们，动不动就堆满了）。丰田及其他汽车制造商发现，这种拉动式系统让整个装配流程变得更快也更便宜了，因为它去除了大量的浪费，也节省了等待时间。事实上，每当发现了某个具体的浪费，它们都能够通过对流程的微调来消除它。

拉动式系统对于开发软件也非常有用，原因并不意外，跟它对制造业有用的原因相同。与其让用户、经理或者项目负责人把任务、特性、请求“推送”给开发团队，不如让他们把这些请求送入一个队列，由开发团队自己从该队列中拉取。当工作发生堆积并在项目中途导致分配不均衡时，他们可以创建一个缓冲区来解决。开发团队在整个项目中可能会用到好几个不同的队列和缓冲区。事实表明，这是一种减少等待时间、消除浪费的有效方法，同时也是帮助用户、经理、产品所有者和软件团队决定开发什么样软件的有效方法。

下面我们举一个拉动式系统如何解决一个熟悉问题的例子：软件团队需要等待所有的特性都写入一个大型规格文档，而后者还必须要经过一个冗长的审核流程。或许该流程为的是征求每个人的意见；也许它不过是那些不敢真正承诺的老板或利益干系人的一种保护自己的手段；或者它干脆就是公司一直以来习惯的做事方法，从来没人想到它其实是一种浪费。如果我们用一个拉动式系统来替换它，会是个什么样子呢？

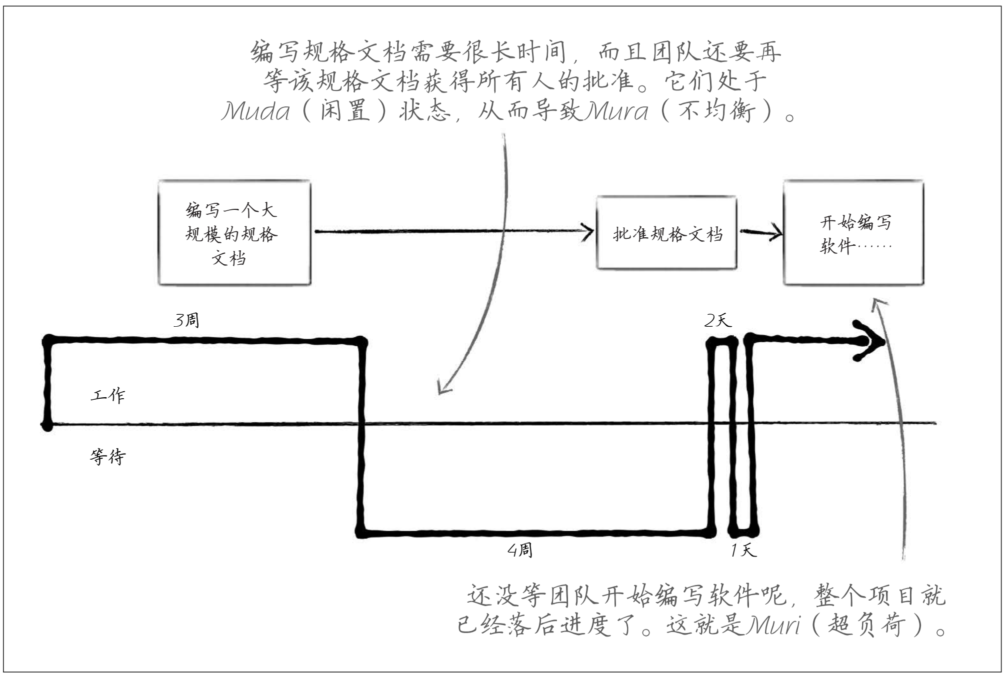


图 8-10：上面这个价值流示意图显示了一个团队等待一个大型规格文档的编写和批准的情况

这是一个很常见的问题，在现实中很多团队都想到了避开它的方法。比如，设计师和架构师可能会基于一个早期的规格草稿和他们对未来的一些预判提前做一些“功课”（正如前文所说的，他们是在“消灭正在入侵的愚蠢”），但是他们能做的毕竟有限。浪费还是会存在，而且不少，尤其是当他们的预判有误而不得不抛弃提前做的部分工作时。

拉动式系统可以很好地消除工作的不均衡并防止团队的超负荷工作。

建立一个拉动式系统的第一步是把工作分解成小型的、可供拉取的块。也就是说，不要编写一个大型的规格文档，而要把系统分解为最小可销售特性，比如一个个独立的用户故事，可能还有针对每个故事的少量文档。这些故事可以单独地进行审批。一般来讲，当一个规格文档审查流程长时间停滞不前时，原因通常是大家对某些特性有不同看法。（发现了吗？把工作分解成小型的 MMF 给予了团队更多的选项。这就是选择思维。）对每个 MMF 进行单独的审批应该至少能够让一部分特性快速得到批准。只要有一个 MMF 通过了批准，开发团队就可以开始进入开发了。这个时候团队不需要去猜测，相反，具体的工作内容已经经过了真正的讨论。可能审批流程确实是有原因的（比如有监管机构的要求，或者确实需要征求每个人的意见）；现在开发团队可以在开始工作之前拿到一个真正充分讨论过的方案。

这就是精益思维的所有部分（着眼全局、找出浪费、用拉动式系统消除不均衡和超负荷）一起共同作用来帮助团队改进工作的方式。但这只是开始。在接下来的一章中，你将学习如何应用精益思维来改进团队开发软件的方式。

要点回顾

- **最小可销售特性**是指团队可以交付的最小的“一块”价值或功能，比如一个用户故事或一个用户请求，就是最终可能进入产品所有者的积压工作表的东西。
- **价值流示意图**是一个可视化工具，它通过展示一个 MMF 的整个生命周期（包括每个阶段工作和等待的时间），来帮助精益团队看到其项目是如何进行的。
- 理解问题的**根本原因**帮助你着眼全局；在这方面，**五个为什么**技巧十分有效。
- 一个可以帮助你的团队做到**尽快交付**的有用工具是**工作进度面积图**，它是一个可视化工具，可以展示出 MMF 如何经过项目的整个价值流。
- 有三种约束你的工作流的重要浪费：Muda（**徒劳无益的事**）、Mura（**不均衡**）和 Muri（**不合理或不可能的事**）。

常见问题

这些信息看起来对运作一个项目是有用的，但是我还是不清楚这对我的日常工作有什么影响。精益会怎样帮助我做好我的工作？

大多数时候，我们不希望听到有人说某个东西“理论化”或者“学究气”，当有人这么说的时候，通常是因为他们没有看到一个马上可以应用的场景。不过，对于精益来说，

这么说很大程度上是没问题的，因为精益本来就是一种思维方式。它并不包含任何可供你每天去做的实践方法。它与敏捷宣言或 Scrum 和极限编程的价值观是一类东西。它同这三个东西一样，是帮助你和你的团队进入正确的思维方式、开发更好的软件并更好地开发软件的极有价值的工具。

在第 2 章中，我们介绍过割裂的视角这个概念，纵观全书，你也已经看到了很多这种割裂所导致的结果：很多团队最好不过是得到了“聊胜于无”的效果，最差则是项目的全面失败。精益思维帮助你从更加宏观的角度看问题，不仅仅是看你的项目，还包括你的整个团队、公司以及规章、政策，还有导致严重的项目问题的文化等。

翻回到本章的开头，再看一遍精益的价值观。这些价值观都是要帮助你把视角从手头的工作上移开，从更高的层次上去审视一切。

一旦你开始寻找浪费，你就会发现它无处不在（消除浪费）。你将不再把开发软件这件事看成是一组零散的任务，而是把它看成一个系统（着眼全局）。你会通过五个为什么这类工具去找出反复影响你的团队的系统性问题，而不是纠结于解决一个个独立的问题（增强学习）。每个精益价值观都会改变你看待项目、团队以及公司的方式。这能够给你一个更加广阔的视角。

在第 9 章中，你将学习如何利用这种视角去实现一些切实的、永久性的改变，来改进你和你的团队开发软件的方式。

你说“Muri”可以翻译成“不可能”，可这难道不是一种悲观的负面态度吗？只要团队有足够的动力，什么事情都是有可能的，不是吗？

要真是这样就好了。这样，我们不妨想象一下下面的情景，看看到底是否存在不可能的事情。

假设你们这个初创公司的 CEO 走进你的办公室，告诉你说你们最大的客户超级喜欢故宫。CEO 把一堆牙签和火柴倒在桌子上，告诉你说，如果你和你的团队能够在该客户到达之前用牙签和火柴制作一个完美的故宫模型，这个客户就会把订单额翻三倍，你们一下子就全都成富翁了。如果模型不够完美，客户就会取消订单，转而跟你们的竞争对手做生意去。那样的话你们的公司只能关门大吉。

客户还有一个小时就要到了。一切在此一举。没有不可能！你肯定能拯救你的公司，对不？

除非你碰巧是一个市政工程师，同时还精通木工艺术，否则你肯定会失败的。有些任务就是没法完成的，不管你多么地充满动力。

有些人管理团队的方式就是，假设要完成任何目标，只要有足够的动力就够了。这种管理方式（我们称之为神奇思维）是很危险的。如果你所处的团队有这种“你无所不能”的风气，那么你肯定会经常面对一些你不可能达到的预期。通常，这些都是因为设置了不现实或不合理的截止时间。但是有些时候干脆就是你需要解决的问题从技术上讲就是不可能的（或者需要极大的工作量才有可能，比如你的代码库状况特别糟糕，到处都是异味）。

这就是 Muri。回顾一下我们前面的定义：不合理；不可能；超出某人能力；过于困难；强制；不得已；强行；过度；无节制。敏捷思维帮助你发现 Muri，同时它还帮助你认识到任何花在不可能的事情上的精力都是浪费。消除这种浪费的最有效方式就是，消除导致你团队去尝试不可能任务的那种神奇思维。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 找出你项目中的所有 MMF。你们是如何管理它们的？你们是否将用户故事写在卡片或即时贴上，然后把它们贴在任务板上？还是在规格文档中对它们有独立的需求说明？找出一个较大的功能或故事，看你是否能把他分解成小块。
- 在你的项目中寻找浪费。把你发现的 Muda、Mura 和 Muri 的例子写下来。
- 从你的项目中那些已经完成的 MMF 中随便拿出来一个，给它制作一张价值流示意图。试试看能否让你们整个团队一起做这件事。然后为另外一个 MMF 也制作一个价值流示意图。这两个价值流之间有什么相似之处？又有哪些不同？
- 找到你的团队经常碰到的一个瓶颈。（在这里价值流示意图会很有用）跟你的团队讨论一下，看看要在未来避免该瓶颈，你们可以做些什么。
- 看一下你和你的团队目前承诺的交付日期。你们是不是真的作出了承诺？有没有其他的选项，可以交付一些不同的东西，但依然兑现你们的承诺？

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 关于精益、精益思维的价值观和价值流示意图：《敏捷软件开发工具》，Mary Poppendieck 和 Tom Poppendieck 著。
- 关于精益和价值流示意图的另外一参考文献：《精益–敏捷项目管理：实现企业级敏捷（修订本）》，Alan Shalloway、Guy Beaver 和 James R. Trott 著。
- 关于 MMF，你可以阅读《用户故事与敏捷方法》（Mike Cohn 著）一书中关于如何分解用户故事的内容。
- 关于选择思维，你可以阅读 *Commitment*（Olav Maassen、Chris Matts 和 Chris Geary 著）。该书是一本关于管理项目风险的图像小说。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 软件团队中的大多数人以前没有遇到过给思维方式起名称的。要帮助他们理解精益是一种思维方式，而不是一套方法，这对于帮助他们接纳精益思维是很有益的第一步。
- 精益思维最为困难的地方之一就是要对谈论浪费感到适应。尽管大多数教练都认同应该让团队保持一个正面、积极的态度，而不是消极、负面的态度，不过就帮助团队学习发现浪费（尤其是对不合理、不可能这两种浪费）这一点来说，开个“批评会”还是有帮助的。
- 要想让针对浪费的讨论保持一个正面的基调，你可以帮助团队成员找出下面这样一些情形：某一事项从“项目”的角度看是浪费，但对于“公司”来讲却是不可或缺的。这可以让他们学会发现浪费，但同时又不会完全负面地看待它。
- 作为教练，你工作的一个部分是避免开发团队与公司之间产生摩擦。如果在公司的文化中，哪怕是对高级经理提出问题也能导致一些不良的后果，那么对敏捷方法的采用就会面临非常严峻的问题。如果可能的话，尽量温和地与经理一起努力，帮助他们意识到他们自己的神奇思维。
- 另外一个保持正面积极氛围的方法是，帮助团队和经理两方把工作流程与该流程中的人区别开来。浪费、低效、反馈，这些都是工作流程相关的东西，而不是对具体的人的判断。

第 9 章

看板方法、流程和持续改进

看板方法本身并不是一种软件开发流程或者项目管理方法。使用看板方法之前，你必须已经具备某种流程或方法，而看板方法的作用就在于逐渐改变你已有的流程或方法。

——David Anderson，《看板方法》

*看板*方法是敏捷团队用来改进流程的一个方法。使用看板方法的团队首先理解它目前开发软件的方法，然后逐渐地对这种方法进行完善。像 Scrum 和极限编程一样，看板方法要求你具备一种思维方式。具体来说，它需要你具备精益的思维方式。在第 8 章我们已经学习过，精益是一种思维方式，它有着自己的价值观和原则。使用看板方法的团队首先把精益思维应用到它的工作中。精益思维为团队提供了一个坚实的基础，这个基础与看板方法结合起来，就为团队提供了改进工作流程的方法。当团队使用看板方法来改进工作流程时，主要专注于从流程中消除浪费（包括 Muda、Mura 和 Muri，即我们在第 8 章中讲过的不均衡、超负荷以及无价值的浪费）。

“看板方法”是一个制造业的术语，由 David Anderson 引入到软件开发领域。David 在 2010 的著作《看板方法》一书中这样描述看板方法与精益之间的关系：“看板方法带来了一套复杂的适应性系统，该系统的目的就是在一个组织中催生出精益的效果。”当然也有一些团队不使用看板方法，也一样把精益思维应用到了软件开发中，但是看板方法是目前为止最常见的，而且，对很多敏捷实践者来说，也是最有效的把精益思维带入一个组织的方法。

看板方法与 Scrum 和极限编程这类敏捷方法的关注点不太一样。Scrum 主要关注的是项目管理：需要做哪些工作，何时交付，以及工作成果是否满足了用户和利益干系人的需求。极限编程的重点则在于软件开发。极限编程的价值观和实践都是围绕着两个重点提出的：第一个是创建一个有利于开发工作的环境，第二个是让程序员养成那些能够帮助他们设计出、编写出简单易维护代码的习惯。

看板方法的核心在于：帮助团队改进其开发软件的方法。使用看板方法的团队对以下几件事情十分清楚：它是如何开发软件的；它与公司的其他部分是如何互动的；它在哪些方面会碰到因无效率和不均衡而导致的浪费；以及如何通过去掉浪费的根源来不断改进。传统上，当一个团队改进其开发软件的方法时，我们称之为“流程改善”。看板方法是通过应用敏捷中的概念（如最后责任时刻）来创建一种简单直接的流程改善方法的例子。

看板方法提供了一系列实践，让你可以稳定住并改善你开发软件的系统。最新的实践集可以在看板方法的 Yahoo! Group（https://groups.yahoo.com/neo/groups/kanbandev/info）上找到。

首先要遵循基本的原则如下所示。

- 从你现在的做法开始
- 愿意追求增量式的、渐进的改变
- 在最一开始，要尊重现有的角色、职责和职位

然后采用以下核心实践。

- 视觉化
- 限制进行中的工作
- 管理流程
- 让流程规则清楚明确
- 实现反馈循环
- 在协作中提高，在实验中演进（使用模型 / 科学方法）

不需要一开始就采用全部 6 项实践。仅采用部分实践的做法称为“浅实践”，并可以随着采用更多的实践和更好的实现逐渐地增加深度。

在本章中，你将学习有关看板方法的内容，包括它的原则、它与精益的关系以及它的实践。你将学到看板方法对流程和队列理论的强调如何帮助你的团队把精益思维应用到实际中，同时你将学到看板方法如何帮助你的团队建立一种持续改进的文化。

故事：有一个正在开发一个手机相机应用的团队
（团队所在的公司刚刚被某大型互联网集团公司收购）

- Catherine——一位开发人员
- Timothy——另一位开发人员
- Dan——老板

9.1 第2幕：紧赶慢赶的游戏

Catherine 和 Timothy 对 Dan 越来越反感。他们两个也清楚截止时间很紧，他们的工作很重要。他们甚至也了解 Dan 身上所承担的交付产品的压力。可是，问题是，好像每一个小项目，不管多小，最后都会进入所谓的“重症监护室模式”（这是 Dan 自己发明的说法，他一开始对下属做微管理时就这么说）。

他们当前的项目也是一样。Catherine 和 Timothy 正在为他们的手机相机应用开发一个新的功能，该功能可以把朋友的照片转换成那种古老的通缉令。这个项目本应是很简单的，就是把他们的相机应用与他们的母公司的社交网络系统集成一下。跟往常一样，他们在测试过程中发现了一大堆 bug，这无疑会导致他们无法按期完工。

“我非常肯定，如果他能放手让我们自己去做事，我们肯定会很顺利地按时完成的。”Timothy 说。

“我明白你的意思，”Catherine 说，“每件小事他都要干涉。说到干涉，现在已经七点了，外面天都开始黑了。还要跟他开进度报告会呢，咱们迟到了。”

现在是“关键时刻”——无法按期完工的时候 Dan 就会这么说。也就是说，任何时候都是“关键时刻”。

Catherine 和 Timothy 走进 Dan 的办公室，坐了下来。他们的队友们已经在了，大家看起来都不太高兴。Dan 已经进入了全面微管理模式。

“注意，我们现在有三个项目处于重症监护状态。Tim、Cathy，从你们的项目开始吧。”

Catherine 说：“我们正在取得进展……”

Dan 打断了她：“你们没有取得进展。项目现在乱七八糟。我已经给了你们足够的时间按你们的方式去做事，现在我们得按照正确的方式来了。”

Timothy 说：“可是有个 bug 必须得修复。”

“怎么总是‘有个 bug’？你们就是不能按期完成我交给你们的项目。现在我必须得插手了，因为你们没有足够的紧迫感。”

“等等，”Catherine 说，“我们总是陷入当前的状态不是偶然的。整个流程中有大量的浪费。”

“什么？浪费？”Dan 问道，“这个提法太负面了。要想成功的话，我们得保持正面的心态。”（Dan 总是强调要保持正面心态，即使是他严厉斥责下属的时候。）

“举个例子来说，比如对用户界面的任何修改都需要每个人同意，这要花掉好几个星期的时间。讨论到一半的时候，你就让我们开始做，结果我们花在修改代码上的时间比最初写这些代码的时间都长。”

Timothy 说：“没错。再比如，我们总是意外地收到 QA 团队的 bug 报告。不知道怎么回事，bug 总是出现，但是我们从来没有花足够的时间去修复它们。”

Dan 看起来生气了。“听着，软件项目就是这样的。不要再指责别人了。说什么是我的问题，是 QA 团队的问题。”

Catherine 受够了：“Dan，够了！”

所有人都看着 Catherine。她以前从来没有这样吼过。

“我们并没有指责任何人。有些问题总是反复出现。像现在这种会议，我们一天开两次，可是都是老生常谈。我们需要讨论那些反复出现的问题。问题一直都是那些问题，可是每次我们都装作很意外的样子。”

Catherine 回敬的这一嗓子让 Dan 吃了一惊。他站起来，瞪了 Catherine 一会儿，然后又坐回到椅子上。“这样吧。你说的那些下次我们再好好考虑。现在，我们得再多花点时间，要多一点紧迫感。现在是关键时刻。”

9.2 看板方法的原则

让我们来仔细审视一下看板方法的基础原则。

- 从你现在的做法开始
- 愿意追求增量式的、渐进的改变
- 在最一开始，要尊重现有的角色、职责和职位

第一个原则（从你现在的做法开始）的焦点与你在本书中学到的所有其他方法都有所不同。

我们花了大量的时间比较敏捷方法与传统瀑布式项目。比如，Scrum 为你提供了一个完整的管理与交付项目的系统。如果你想要采用 Scrum，需要设置新的角色（Scrum 主管和产品所有者），而且还需要给团队增加新的活动（冲刺计划、每日站立会议和任务板等）。这些是采用 Scrum 时必要的做法，因为它是一个管理项目和交付软件的系统。

看板方法本身并不是一个管理项目的系统。它是一种方法，用来改进你的流程，即你的团队开发和交付软件的那些步骤。在谈到“改进”某个东西之前，你需要有个出发点，而看板方法的出发点就是你现在的做事方法。

9.2.1 找到一个出发点并由此进行实验性的演进

习惯性问题的棘手之处就在于，它已经融入了你的习惯之中。

当你的团队做了某件事并最终导致 bug 或者耽误工期时，在做这件事的那一刻，这看起来并非是一个错误。事后你尽可以进行充分的根源分析，可是，再次面对相同的选择时，你的团队很可能还是会作出相同的决定。这是人的本性。

比如，假设一个开发团队发现，每次把软件交付用户之后，用户总是会提出他们找不到所要的功能。如果说开发人员太过粗心，总是会忘记开发与用户讨论过的一两个功能，这当然是有可能的。但是更可能的情况是，团队在收集用户需求或者与用户沟通时存在某种反复出现的问题。

流程改进的目标就是找出这些反复出现的问题，弄清楚这些问题的共同之处，并采用必要的工具来克服它们。

这里的关键在于第二步：弄清楚这些问题的共同之处。如果你简单地假定某个开发人员就是记不住用户要求的那些功能，或者用户总是不断地改变主意，那么你实际上相当于认为问题是无法解决的。但是，如果你认为有某个真正的根本原因反复地发生，那么就有找出并克服该问题的机会。

这正是看板方法的起点：审视当前的工作方式，并把这看作一组可以改变、可以重复的步骤。看板方法团队把一直遵循的步骤和规则称为*策略*。说到底，团队成员需要意识到自己的习惯，看到自己每次开发软件时都采取了哪些步骤，并且把这些都落到纸面上。

有时候要把一个团队所遵从的那些规则都写出来并不是件容易的事，因为很容易会走入误区，变成基于项目的成果对一个团队（或某个团队成员个人）的评判：如果项目成功，团队中的每个人肯定都是好样的；如果项目失败了，那他们一定是不得力的。这样做不公平，因为这假定了项目的一切都在团队的控制之内。精益思维帮助我们避免进入这种误区，方式就是着眼全局，在这里也就意味着要意识到存在着一个更大的系统。

值得再次强调的是：每个团队都有一个开发软件的系统。这个系统可能是混乱的。它可能经常改变。它可能仅存在于团队成员的头脑中，却从未被遵循它的人们摆在桌面上讨论过。对于遵循像 Scrum 这类方法的团队来说，该系统有章可循，而且人人都能理解。但是很多团队的这种系统还处于一种“原始部落”的状态：我们一直是在项目开始时跟这几个客户代表沟通，或者制订那样的日程表，或者编制故事卡片，或者在与经理开过一个小会之后，马上让程序员介入并开始编码，等等。

这种系统就是看板方法的起点。团队已经有一个运作项目的系统了。看板方法所要求的，就是团队成员要理解该系统。这就是“从你现在的做法开始”的具体含义。看板方法的目标是对该系统做小幅度的改进。这就是“追求增量式的、渐进的改变”的含义，这也是看板方法有着*在协作中提高，在实验中演进*这一实践的原因。在精益思维中，量化指标是着眼全局的一部分，而量化指标正是实验和科学方法的核心。看板方法团队会从其软件开发方式开始，测量量化指标来理解它。之后，团队成员会作为一个团队一起来做出具体的改变（在本章后面，你会学习这些改变到底是怎么起作用的），并且查看量化指标来确定这些改变是否取得了他们想要的效果。

在让你的团队所使用的开发方法不断演进这一点上，增强学习这项精益价值观也是很重要的一部分。贯穿本书，你已经学习了有关反馈循环的内容。当你与他人协作来测量你的系统并实验性地进行演变时，反馈循环就成了收集信息并将其反馈给系统的一个非常重要的工具；看板方法的*实现反馈循环*这一实践对你来说应该不难理解，它也应该能够帮助你看到看板方法与精益的紧密联系。

对于看板方法的另外一个原则（在最一开始时，尊重现有的角色和职责）来说，增强学习也是一个需要纳入考虑的因素。比如，一个团队总是以一个项目经理、一个业务分析员和一个程序员之间的一次会议开始。可能并没有成文规定说在这个会议上应该讨论什么，但是仅仅通过刚才列出的几个头衔，你可能就能大致猜出会议上会讨论哪些东西。这正是看板方法尊重现有角色、职责和头衔的原因之一，因为它们是系统的一个重要部分。

这些看板方法原则的一个共同特点是：它们起作用的前提条件是，开发团队必须花时间去理解它们自己开发软件的那个系统。如果存在一种正确的开发软件的方法，那大家就那么干了。但是我们在本书的第 2 章中就提到了，并不存在银弹（即，不存在一组“最好的”时间能够保证团队每一次都能够成功地开发出软件）。即使是同一个团队，使用同样的实践，也可能在一个项目上取得成功，却在下一个项目上败走麦城。这就是为什么看板方法要从理解当前运作项目的系统开始：一旦你看到了整个系统，看板方法就能够为你提供改进它的方法。

等一下，这么说看板方法并不能告诉我如何运作项目？

没错，它不能！[1] 看板方法要求你先理解你当前运作项目的方式。它可能是 Scrum、极限编程、“聊胜于无”的 Scrum、有效率的瀑布式流程、无效率的瀑布式流程，甚至可以是杂乱无章、跟着感觉走的做事方法。一旦你弄清楚了你的团队现在是如何开发软件的，看板方法就可以为你提供改进它的方法。

那么，如果你已经有了一种开发软件的方法，你为什么还需要看板方法呢？

大多数团队都是能够交付出东西来的。可是，你怎么知道：一个团队是否浪费了大量的时间和精力？团队是否在做那些创造很多价值的事情？团队成员是否被要求去以某种方式工作，但这种方式会习惯性导致问题或者给交付有价值的软件制造困难？

当我们已经有了一套系统的时候（不管这个系统是什么样的），我们大部分人都不太会去质疑它。即使你已经在使用 Scrum 或极限编程，你还是有可能浪费大量的时间和精力却不自知。习惯性问题非常难以发现。每个人都可能在遵守规则，做正确的事情。但是就像行为可以从一个系统中生发出来一样，浪费也可能在多人协作的环境中产生。

我们在第 8 章中已经看到了一个例子，一个团队发现其交付时间很长，尽管每个人都在不间断地工作，没有人偷懒或者等待工作。可是，就算每个人都不间断地工作，对于软件的用户来说，却存在着很大的延迟，而且团队中没有人发觉，因为团队成员感觉他们已经在竭尽全力地尽快完成工作了。

看板方法解决的正是这类问题。

9.2.2 用户故事进去，代码出来

> 系统性思维把组织看成是系统；它分析组织的各个部分是怎样相互关联的，以及组织是如何作为一个整体随着时间的推移而运转的。
>
> ——Tom Poppendieck，Mary Poppendieck，《敏捷软件开发工具》

改进一个系统的第一步是要意识到它的存在。这是着眼全局这一精益原则背后的思想。当你着眼全局的时候，就不会再认为团队是在做一连串独立的、互不相关的决定，就会开始思考它是在遵从某个系统来做事。在精益中，这称为*系统思维*。

每个系统都会接受一些输入，然后把它们转化为输出。从系统思维的角度看，一个 Scrum 团队是什么样的？你可以把 Scrum 看成是这样一个系统：它把项目的积压工作表条目作为输入，最终产生代码作为它的输出。很多 Scrum 团队的项目积压工作表完全由用户故事组成；这些团队完全可以把它们自己当作是把故事转化成代码的机器。

很显然，它们是团队而不是机器，而且我们当然不想陷入那种把人当作机器或齿轮的坏习惯。不过，把你所做的工作作为一个更大系统的一部分来进行思考是有它的价值的。如果

注 1：看板方法不是一种项目管理方法，但这可不是说看板方法对项目经理就没有用！事实上，David Anderson 写过一个博客系列，专门解释项目经理在看板方法中的角色，他还在 Lean Kanban University 开设了名为“用看板方法进行项目管理”的课程。关于这个主题，我们推荐您阅读他的博客文章（http://www.djaa.com/project-management-kanban-part-1）。

你把系统思维应用到你的 Scrum 团队中，它能够帮助你更容易地发现什么时候你所做的不会直接地（或者甚至不会间接地）帮助把故事转化成代码。通过意识到 Scrum 是一个系统，你就能够理解它怎样才能工作得更好，并对它进行改进。这才是能够带来改进的思维，就像第 5 章中 Jeff Sutherland 对每日站立会议中那些问题的改动那样。那是一个把系统思维应用到 Scrum 中并带来增量式的、渐进的改进的很好例子。

看板方法要求你从理解你和你的团队正在使用的系统开始。即使你没有使用一种叫得上来名字的方法，也依然可以通过应用系统思维来弄清楚你们是怎么工作的。

1. 每个软件团队都有一套系统，不论成员是否知道

一个 Scrum 团队有它的一套系统，这很容易理解。可是如果你的团队没有类似这样的一套系统呢？可能你就是简单地上来就开始开发软件而已。感觉上你肯定不是每次都用同一种方法来运作你的项目。那么说，你是不是就没有一套系统呢？

人（尤其是团队中的人）有一个很有趣的特点，那就是我们总是遵守一定的规则。这些规则未必是纸面上的，而且如果不存在这样的规则的话，我们常常会自己编造出一些规则。不过，人类一直以来都通过直觉感知到规则，而且一旦某个规则进入我们的头脑，它就很难再被撼动。而且，即使你觉得你没有遵守什么规则，当有新成员进入你的团队时，你肯定会在那个人打破某项不成文的规则时注意到这一点。

精益甚至给我们提供了把不成文的规则转换成一个系统的工具：价值流示意图。当你拿过一个 MMF（即我们在第 8 章中学习的最小可销售特性，比如故事、需求和用户请求等）并将其从开始到变成代码的路径画成一张价值流示意图，你就已经把你的系统中的一个路径落实到了纸面上。

当你在一个有很多不成文规定的团队中工作时，一个 MMF 很可能与另外一个 MMF 所经过的路径不一样。可是因为人会自然地凭直觉感知并遵守规则，你很可能只需画出少量的几张价值流示意图，就可以覆盖你的团队中涉及的大部分 MMF。如果你能做到这一点，那么就能够构建出一个你和你的团队所遵从的系统的准确描述。该系统的第一个步骤就是确定一个 MMF 应该进入那一条价值流。

创建一个对所有人都以相同方式运作的系统是值得的，哪怕该系统中对一个特定 MMF 有很多条不同的可能路径。一旦你理解了这个系统的工作方式（换句话说，一旦你看到了全局），就能够开始真正知道哪些路径是浪费的，并对你的系统作出增量式的、渐进的改变。

2. 看板方法不是软件开发方法，也不是项目管理系统

人们学习看板方法的时候最常见的误区是把它当作一个软件开发的方法。它不是。看板方法是一种流程改进方法。它帮助你理解你所使用的系统，同时帮助你找出改进的方法。

你可以花一两分钟翻到本章几页后的内容浏览一下。你会看到一些像是任务板一样的图片。它们并非任务板。它们叫作看板。你之所以知道它们不是任务板，是因为它们上面没有任务，只有工作项（work item）。一个工作项是一个单独的、自包含的、可以在整个系统中跟踪到的工作单元。它一般比一个 MMF、需求、用户故事或其他独立范围的单元稍大一些。

任务板与看板的一个区别在于，虽然任务会在任务板上流动，但工作项并非任务。任务是

为了让工作项在系统中推进而需要做的事情。换句话说，任务就是推动工作项前进的机器当中的齿轮。因此，你可以使用系统思维来理解你的软件开发工作流，却不需要把人当作机器的齿轮。任务才是齿轮；人依然是有着各自性格、需求和动机的独特个体。

看板上的那些列可能看着与价值流的步骤很类似；但是，很多看板方法的实践者会把价值流示意图与看板区分开来。他们会把绘制工作项状态的过程与价值流分开做，他们把这个过程叫作*绘制工作流*。这里的不同之处在于，绘制价值流是一个精益思维工具，用于帮助你理解你所在的那个系统；而绘制工作流则是看板方法如何确定每个工作项需要经过的那些实际步骤。（你会在本章的后面学习如何创建一个看板。）

下面我们来举一个例子，说明任务板与看板在功能上的不同。Scrum 的重点在于帮助团队进行自组织，并达成团队的共同承诺。当用到任务板的时候，团队已经选择了要包含在冲刺中的积压工作表条目（即工作项），并把它们都分解成了具体的任务。随着任务在任务板上的流动，工作项也从“待处理”栏向着“已完成”栏移动。对团队来说，它能够感觉到项目在向前推进。

一般的看板只包含那些更大的工作项，而不包含细分出来的单个任务。任务板只能“看到”任务项是“待处理”“处理中”还是“已完成”，而看板反应的则是一个更宏观的视角。工作项是从哪里来的？产品所有者怎么知道哪些工作项要放入项目的积压工作表，如何排列优先级？团队完成一个工作项之后，有没有一个生产团队来确保它进行了正确的部署？看板反映了工作项的更高层面的生命周期，其范围超出了开发工作本身。看板的某些栏目会包含开发团队接手该工作项之前和之后该工作项需要经过的步骤。

此外，看板还能够帮助发现那些不会出现在任务板上的问题。很多 Scrum 团队非常善于开发它们被要求开发的软件，但是它们还是会发现自己的工作让用户失望了。可能是它们开发的那些工作项不是那些能够满足用户需求的项目。或者可能在团队开发那些工作项的之前和之后，存在着很长的延迟，也许是因为很长的审查或者部署流程。尽管这些问题完全不在开发团队的控制之内，它们却常常为此而获咎。看板在这里会很有帮助。

因此，尽管看板方法并非一个项目管理系统，但是它与团队的项目管理方法有着很重要的关系。看板方法是要改善和改变项目的现有流程，而这能够、也一定会影响项目的管理方法。一般看板方法被用来改进流程的可预测性，而这将会影响到项目的计划和排期。对看板方法及其标准的更广泛应用很可能会对项目管理的方法产生显著的间接影响。[2]

在 9.3 节，你将学习看板方法的各项实践、如何创建看板，以及使用看板来对整个系统进行增量式的、渐进的改进。

9.3 用看板方法改进流程

我们已经知道看板方法是一种专注于过程改善的敏捷方法，它基于的是精益价值观和精益思维。看板方法是由 David Anderson 提出的，他在微软和 Corbis 工作的时候最先开始了对精益中的思想的一些实验。与精益的很多内容一样，“看板”这个词也源于日本汽车制造商

注 2：感谢 David Anderson 在本段的用词上对我们的帮助。

的创造。可是，看板方法到底“敏捷”在哪呢？它与传统的过程改善方法有什么区别呢？

自打人们开始开发软件以来，软件团队就一直在进行过程改善了。理想情况下，过程改善应该效果很好：团队为此得到非常好的支持，它们采集量化指标，找出问题，做出改进，然后重新来过，寻找更多可改进的地方。最后，这些改进帮助整个组织首先做到将过程可重复化，然后把过程管理起来，最后将其纳入统计意义上的控制。有大量的公司在这方面取得了广泛的成功。

如果你是一个曾经努力进行过过程改善的开发人员，在读了上面这段话之后，可能会充满挫败感地想要把本书放下了。

“过程改善”这个词常常让人脑海中浮现出类似漫画 *Dilbert* 中的画面：无穷无尽的委员会、塞满了过程文档的活页夹，等等。这是因为一般的过程改进与理想中的有很大区别。在一般的过程改进努力中，一个大公司认为他们的程序员没有能够高效率地产出软件（或者他们因为合约或销售的原因，需要一个过程认证证书），于是他们雇佣一个顾问公司来花费大量的时间（还有金钱）绘制一大堆现有流程和想要的流程的图表，同时对团队进行培训，让团队使用新的流程。然后，开发团队花上 10 分钟时间尝试一个新流程，发现它不自然、别扭、难以适应，然后就再也不用它了。可是，因为是上司提出来做这种过程改进，团队还是要做做样子表示表示，所以团队会按照新流程的要求填写各项表单（比如范围文档和工作陈述）、撰写强制性的文件（比如几乎是空白模板的代码审查会议记录和测试报告）。每一次成功的过程改善尝试背后（是的，还是有那么一些成功的）都有大量不那么成功或者完全失败的努力，而后者的一个副产品则是人们对于“过程改善”这个词的深恶痛绝。

看板方法与传统过程改善有一个很大的不同。传统过程改善中，决策通常是由高级经理发起，由一个委员会（比如一个软件工程过程组）拍板，再通过开发团队的上级传达给底下的人。而看板方法中，这种改善则交到了开发团队的手上，而这也正是敏捷团队在看板方法上取得了成功的原因之一。团队成员自己找出他们工作流程中的问题，提出他们自己的改善建议，量化效果，而且自己对自己负责。

那么，看板方法如何帮助一个团队改进其流程呢？

9.3.1 将工作流程可视化

改进一个流程的第一步是理解团队目前是如何工作的，而这正是看板方法的可视化实践的目的所在。这听起来简单，做起来可不容易，这也是很多传统过程改善做法容易出错的地方。

想象一下假如你是一个程序员，你的上司过来问你：“你是怎么开发软件的？”你的任务是把你如何工作的过程写下来。于是你打开 Visio（或者 Omnigraffle，或者其他什么画图软件）并开始把你每天做的所有事情画成一张流程图。这时你意识到，尽管大家嘴上都说要做代码审查（或者在提交代码前先进行测试，等等），可是你其实并不是每次都做。但是你脑子里想的是，代码审查是一个好主意，你肯定有时候会做的，于是你就把它加到你的流程图里了。这是人的本性。你很容易就能说服自己把它加到流程图中，如果这种做法很好，那么把它写出来可能会帮助你留意每次都这样去做。

这种做法会把你的过程改善努力彻底搞砸。

一个原因是这种做法会掩盖真正的问题。如果说你加到流程图中的那个步骤是一个好主意，那么现在它出现在了流程图中搞得好像你已经在这么做了似的。这样别人就不会想要去问："我们为什么不这么做呢？"他们怎么会这么问呢？你明明白纸黑字写着你已经在这么做了呀！问题是，你不是每次都这么做，如果这背后有某种原因呢？比方说，代码审查总是被取消，因为只有团队的资深成员才被允许审查代码，而他们又总是很忙。你们将无法发现这个原因，更谈不上尝试解决这背后的问题，因为每个人都会看到流程图上明明写着代码审查一直都在做，于是转而去关注其他可以改进的地方了。

在看板方法中，可视化意味着把团队的做法原封不动、毫无保留地落实到纸面上，不经任何的美化。这是精益思维的一部分：使用看板方法的团队对待精益中的"着眼全局"这一原则是非常严肃的。如果团队的思维方式是正确的，在把工作流程可视化的过程中对它进行调整就是不对的，因为那会妨碍你看清全局。"尽量推迟决定"这一价值观在这里也很重要：你还没有获得你如何开发软件的所有信息，所以关于如何改变你的工作流程的决定应该留待一个更晚的时间去做。

像其他敏捷方法一样，使用看板方法的实践能够帮助你进入精益的思维方式并真正接受精益思维。你越是能够准确客观地把工作流程可视化，你就越能把"着眼全局"和"尽量推迟决定"锲入你自己的思维当中。

那么应该如何将工作流程可视化呢？

用看板将工作流程可视化

*看板*是一个用于将团队工作流程可视化的工具。[Kanban（看板方法）中的 K 一般大写；而 kanban board（看板）中的 k 一般小写。] 一个看板跟 Scrum 中的任务板很像：它一般包含一些画在白板上的栏目，每个栏目里面贴着一些贴纸。（看板上更常见的是贴纸，而不是索引卡片。）

任务板与看板之间有着三个非常重要的不同。你已经了解了第一个不同，即：看板上只有故事，而没有任务。第二个不同是看板上的栏目通常根据团队的不同而不同。最后，看板每一个栏目中的工作量可以设限制。关于这些限制我们后面会讨论；现在，让我们先专注于看板上的栏目，以及为什么不同团队的看板栏目会有所不同。一个团队的看板上可能分成了我们熟悉的"待处理"、"处理中"和"已完成"三个栏目，但是另一个团队的看板上可能是完全不同的栏目。

当一个团队想要采用看板方法时，所要做的第一件事情就是通过创建一个看板来把它的工作流程可视化。比如，在 David Anderson 所著的《看板方法》一书中，最开头部分有一个看板，看板上的栏目是：输入队列、分析（处理中）、分析（已完成）、可开发、开发（处理中）、开发（已完成）、可构建、测试和可发布。使用这个看板的团队的工作流程可能是这样的：每个特性都需要经过分析、开发、构建和测试这几个环节。所以，团队可能会从类似图 9-1 那样的一个看板开始，在各栏目中贴上贴纸来代表经过系统中的各个工作项。

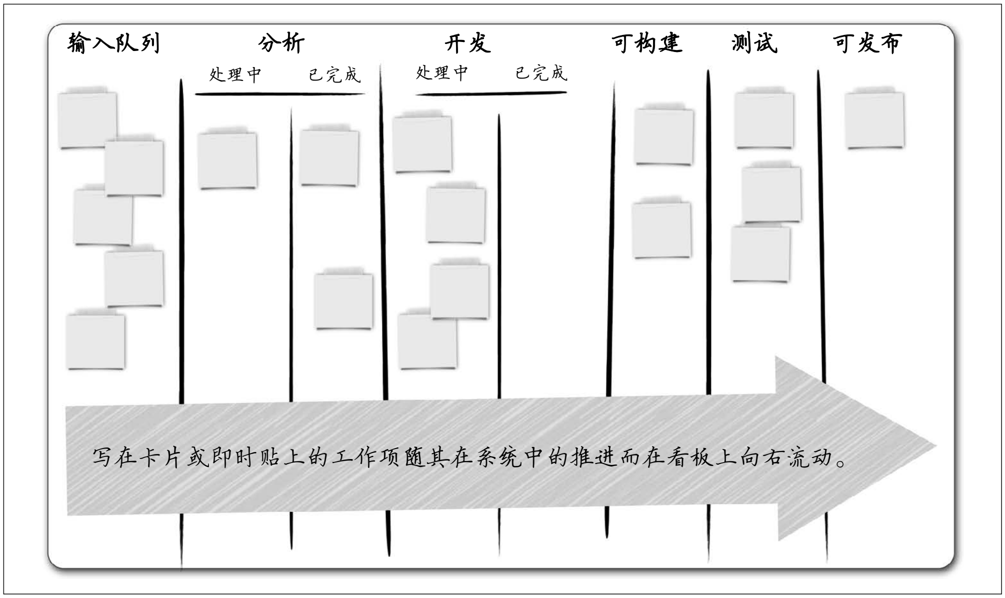


图 9-1：写在即时贴上的工作项在一个看板上流动的例子。David Anderson 在他的《看板方法》一书中用到了含有图中栏目的看板，不过根据团队中人们工作方式的不同，看板可以有与上图不同的栏目

开发团队接下来就像 Scrum 团队使用任务板那样来使用看板。看板团队会召开所谓“过看板”的会议（一般是每天一次），会上团队成员会讨论看板上所有工作项的状态。看板应该已经反映了系统当前的状态：每个完成了当前步骤的工作项应该已经被推进到下一个栏目，也就是把它的那张即时贴揭下来，贴到下一个栏目中去。不过如果这些还没有做，开会时大家会先保证看板的状态是最新的。

需要理解的很重要的一点是，看板把团队所使用的流程可视化了。这里或其他书中（比如 David Anderson 的《看板方法》）所给出的例子不过都是实际场景中的例子。一般来讲，你绝不该照抄别人的看板；相反，你应该通过研究你自己的工作流程并将它可视化来制作你自己的看板。不考虑实际场景地照抄别人的过程定义的做法与看板方法的演进式改变的思想是相悖的。如果看板方法明确要求你以你现在的做法开始，那你就不应该从照抄别人的做法开始。[3]

让我们回到第 8 章中的那个例子：一个团队正在尝试应对过长的交付时间和失望的客户。如果你翻回到最开始该团队的描述，那基本上就是项目经理描述给老板的初始工作流程。我们来快速回顾一下。

(1) 团队从用户处得到一个请求
(2) 项目经理为下一个发行版本安排特性
(3) 团队开发该特性

注 3：感谢 David Anderson 在本段的用词上对我们的帮助。

(4) 团队测试该特性
(5) 项目经理确认测试能够通过
(6) 特性完成并被包含在下一个发行版本中

第 8 章以文字段落描述这个工作流程，而上面的有序列表则是另一种描述方式。与这两个相比，可视化在描述工作流程上更加有效。图 9-2 展示了这个项目经理口中所谓的“阳光大道”版工作流程在看板上的样子。

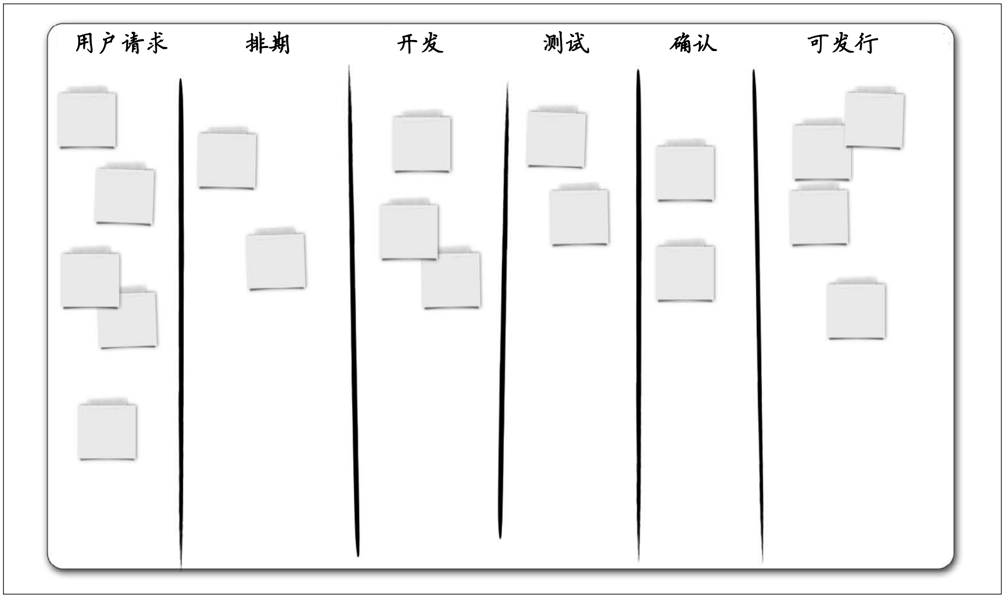


图 9-2：这是每个人以为的项目的推进方式

不过，现实生活中可不会一路畅通。在第 8 章中，开发团队用“五个为什么”深入了解工作流程。这之后，该工作流程变成了下面这样。

(1) 团队从用户处得到一个请求
(2) 项目经理为三个月后的发行版本安排特性
(3) 团队开发该特性
(4) 团队测试该特性
(5) 项目经理确认测试能够通过
(6) 项目经理安排一个针对高级管理层的演示
(7) 如果高级管理层有人想要让团队对该特性做些修改，项目经理会对这些修改做影响分析，同时该特性回到第 1 步；而如果不需要做修改，则该特性继续执行第 8 步
(8) 特性完成并被包含进下一个发行版本中

现在我们知道存在一个额外的步骤，即高级经理可以在团队以为它已经完成某个特性之后，再有选择性地对该特性进行修改并把它推迟到未来的某个发行版本中去。

为了把这种更好的理解表示出来，我们在看板上为那些等待向高级经理演示的特性增加一个叫作“经理审核”的栏目。

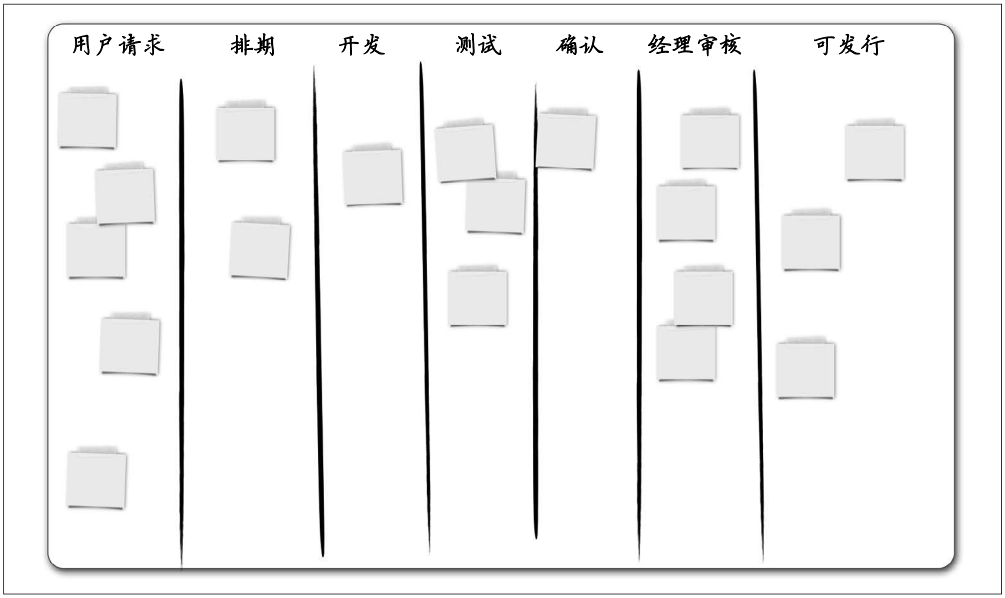


图 9-3：这个看板对项目运作的呈现更准确，也更现实

现在我们有了团队工作流程的一个更加准确的可视化结果。如果我们让看板经历一个发行周期，问题的所在就显而易见了。工作项会在“经历审核”这一栏里面形成堆积，并且会一直堆积在那直到临近发行的时候，如图 9-4 中所显示的那样。

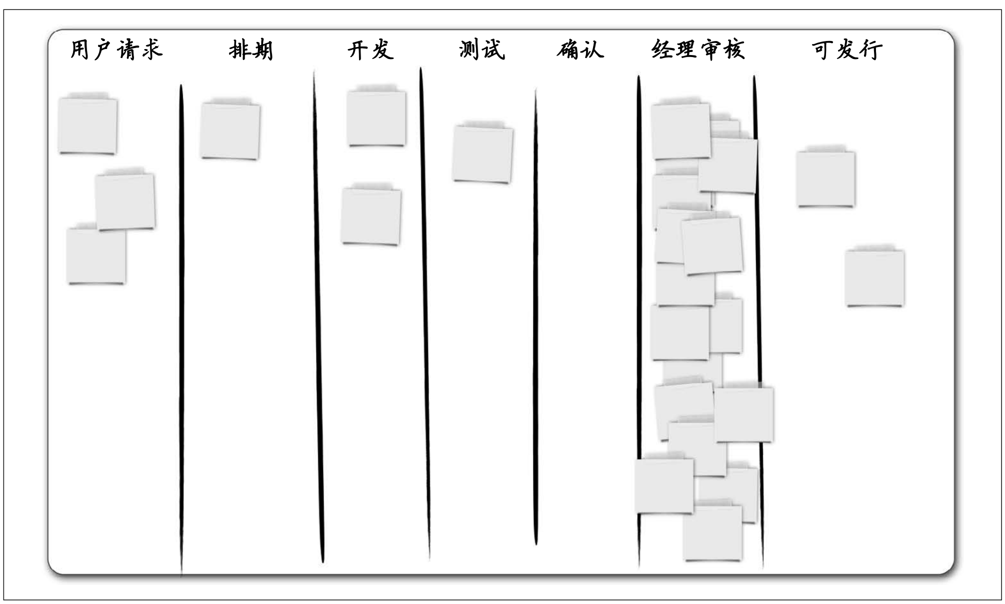


图 9-4：当你使用看板来把工作流可视化时，因不均衡（Mura）导致的问题就变得容易发现了

可是，那些为了给经理提出的修改腾地方而被推迟到未来的发行版本中的那些工作项呢？我们特别关心那些工作项，因为它们是导致用户流失的原因。有时候针对那些工作项的工作已经开始了，即使推迟到未来的版本中，也还是要继续做。当工作项在经理审核后被推迟了，它们就会被打回到流程的开始阶段。让我们来保证这种情况能够在看板上表示出来，我们可以在看板的最开始增加一栏，叫作“被经理推迟”，然后把被推迟的工作项的贴纸挪过来（我们在这些贴纸上点上一个小圆点，以便在这些被推迟的工作项第二次经过整个看板的时候能够容易地被识别出来）。

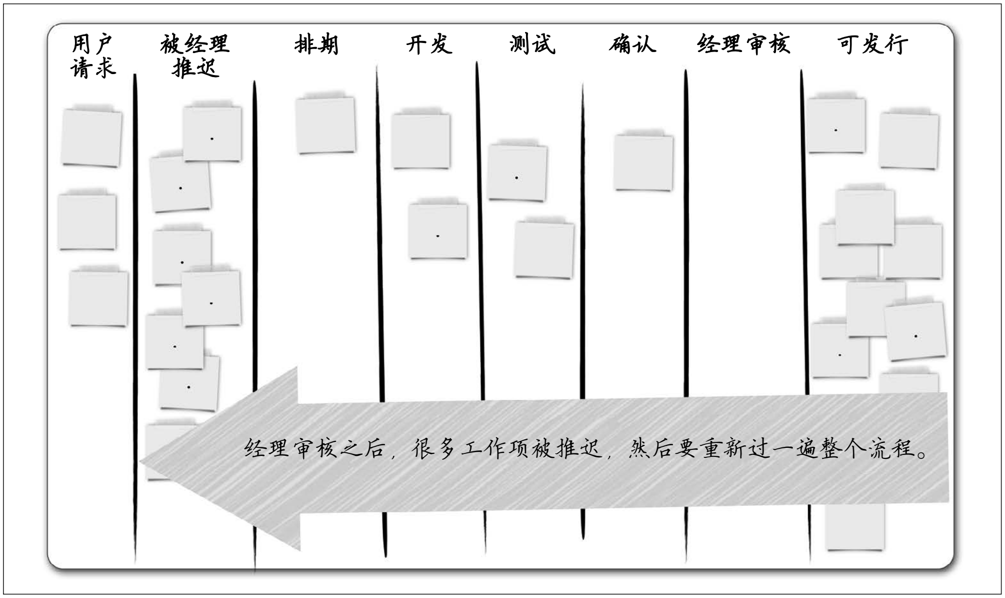


图 9-5：看板让流程中的浪费更加明显，因为你可以看见即时贴会多次经过整个流程

这是对该团队的流程的一个很好的可视化。现在我们能够清楚地看到项目的哪个环节出了问题，以及为什么交付时间一直那么长。团队中可能有人已经清楚问题的关键了，不过，有了看板，任何人都可以通过查看看板来弄清楚问题的根源。更重要的是，看板成为了客观的、显式的证据，证明高级经理审核特性的方式是过长交付时间的一个主要原因。

9.3.2 限制进行中的工作

> 我们绝不会让机房里的服务器满负荷运转，在软件开发上我们为什么没有认识到这一点呢？
>
> —— Mary Poppendieck，Tom Poppendieck，《敏捷软件开发工具》

一个团队在特定时间内就能做这么多工作。我们在 Scrum 和极限编程中都学到过这一点，它也是精益思维的一个重要部分。当一个团队同意在某个时间之前完成它能力之外的工作量时，糟糕的事情就会发生。它要么无法交付一部分工作，要么交付的产品质量有问题，要么以一种不可持续的节奏工作并给未来的发行版本带来很高的代价。而且，有时候一个

团队承受了比它能够胜任的更多的工作量这一点并不是显而易见的：每个独立的截止时间可能看起来都比较合理，但是如果每个人都需要同时在多个项目间或任务间进行多任务处理，那么团队就会慢慢变得负荷过重，而任务切换所需的额外精力可能会占去它一半的生产力。

把工作流程可视化能够帮助团队清楚地看到这种超负荷的问题，而这正是向最终解决该问题迈出的第一步。当看板上的即时贴在一个栏目中堆积时，不均衡和超负荷（我们在第 8 章中讲到过）的问题就变得很清楚了。幸运的是，队列理论并不仅仅提醒我们注意这个问题，它同时也为我们提供了解决该问题的方法。一旦在工作流程中发现了不均衡，我们可以使用队列理论来控制流经整个系统的工作量，具体做法是，为允许堆积的工作量设定一个上限。这就是看板方法中的*限制进行中的工作*实践。

“限制进行中的工作”的意思是，为项目工作流程中的每一个容器中可以存在的工作项的数量设定一个上限。很多人都倾向于努力让工作项在工作流程中尽快地向前推进。而对于单个工作项来说，工作流程是线性的：如果你是一个开发人员，完成了一个特性的设计，而工作流程中规定的接下来的步骤是开发该特性并交给测试人员测试，那么你接下来要做的就是编写该特性的代码，你很容易会专注于开发该特性并尽快交给测试人员，因为你刚刚完成了这个特性的设计工作。

可是，如果测试团队已经在超负荷运转了呢？这种情况下，马上开始该特性的开发工作就不合时宜了，因为开发完了之后它也不过是放在那里等着，因为没有多余的人手马上测试它。这将导致测试团队负荷过重。那么该怎么办呢？

对此，看板方法回到了精益思维，具体地说，是回到了你在第 8 章中学习过的选择思维。看板团队使用看板的一个原因是因为它能够展示出你所有的选项。如果你是那个刚刚完成了一个特性的设计的开发人员，你很可能会认为你现在*承诺*了接下来要开始编写该特性的代码。但是给那个特性编写代码只是一个选项，而不是一个承诺。当你看一看整个看板时，你会看到很多你接下来可以做的即时贴。也许在前面的栏目里有其他特性需要设计，或者在后面一点的栏目中有一些特性中的 bug 被测试人员发现了并且需要修复。事实上，大多数时候你都有很多选项可以选择。那么你选择哪一个呢？

给你的工作流程的每个步骤设置一个进行中工作的限制意味着限制了可以进入该步骤的特性的数量。这样可以帮助限制团队的选项，从而让团队的选择变得更容易，同时还能够避免超负荷问题，并保持特性在工作流程中能够尽量高效地流动。比如，当你完成了一个特性的设计工作时，你看到工作流程中编写代码的那个步骤已经达到了预设的上限，那么你就会去寻找其他可做的选项，这样测试团队就不会负荷过重了。（花一分钟想一想：你是否能够看出这种做法是怎样减少特性的平均交付时间的？如果你能看出来，那么你已经开始能够灵活运用系统思维了！）

让我们回到前面的那个例子，某个团队用五个为什么方法找出了其交付时间长的根本原因。一旦我们为该团队创建一个看板，超负荷的问题就变得很清楚了：帖子会在“经理审核”这一栏堆积起来。因此，为了限制该团队工作流程里进行中的工作，我们只要想办法给堆积的特性设置一个严格的上限，达到该上限时经理就需要开一次审核会议。

在看板方法中，一旦你意识到了工作流程中的这类问题，处理它的办法就是给它*设置进行*

中工作上限。具体做法是：团队需要与其上司和其他高级经理开会并说服后者同意一个新的规则。新规则会给进行中的工作设置一个上限，即看板的“经理审核”一栏中只能存在一定数量的特性。看板和交付时间指标给团队和其项目经理足够的客观证据来说服经理同意这样的安排。

当我们给某一栏设置这样的限制之后，它就不再是堆积工作项的货场，而变成了一个队列，你工作流程中的这个步骤里的工作项被流畅、有序地管理起来了。

至于进行中工作的上限应该设成多少，并没有一定之规；团队会采用一种演进式的方法来设置这个上限。看板团队一般会先选择一个大家都同意的合理数值，然后通过量化指标在实验中调整。对于我们例子中的团队，我们不妨假设每个发行版本一般包含 30 个特性，而高级经理觉得他们可以在一个发布周期中会面三次，那么我们就选择 10 作为进行中工作上限。我们会在看板的“经理审查”一栏里面写上 10 这个数字，如图 9-6 所示。

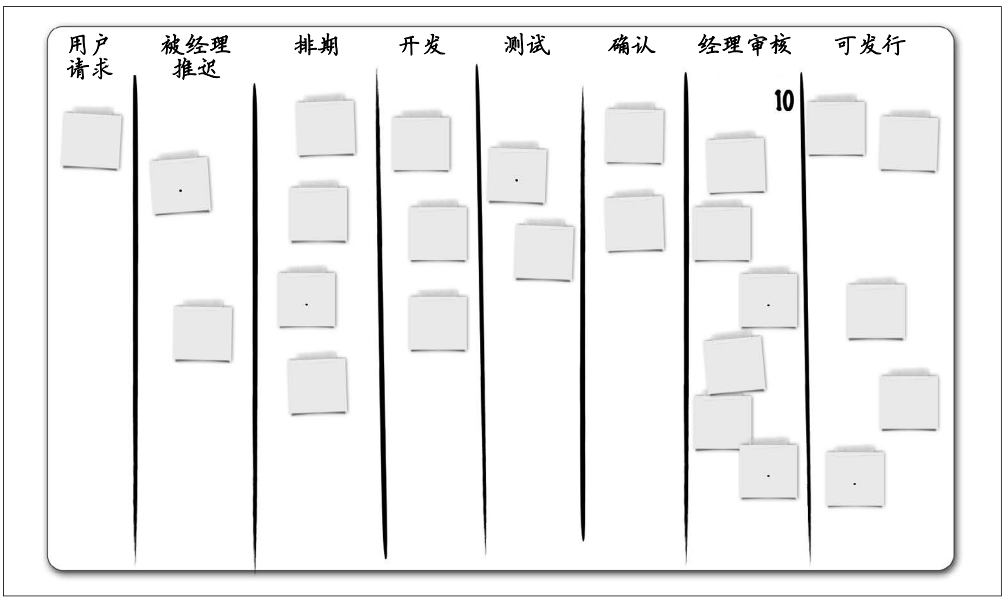


图 9-6：经理审查一栏中的 10 代表该栏的进行中工作上限

当第 10 个工作项进入经理审查一栏并导致它达到其上限时，会发生什么呢？上限达到之后，开发团队就不再把其他工作项推进到那一栏了。开发团队应该还会有其他工作可做。如果开发工作已经全部完成，那么开发团队可以帮助 QA 团队测试软件。也许开发团队还有一些可以处理的技术债务，看板上应该有技术债务相关的贴纸。团队唯一不会去做的一件事就是把更多的特性推进到“经理审核”一栏。这是他们与高级经理达成的协议：该栏一旦达到上限，高级经理就应该举行审核会议，如果他们不这么干，那么工作就会一直堆在那里。

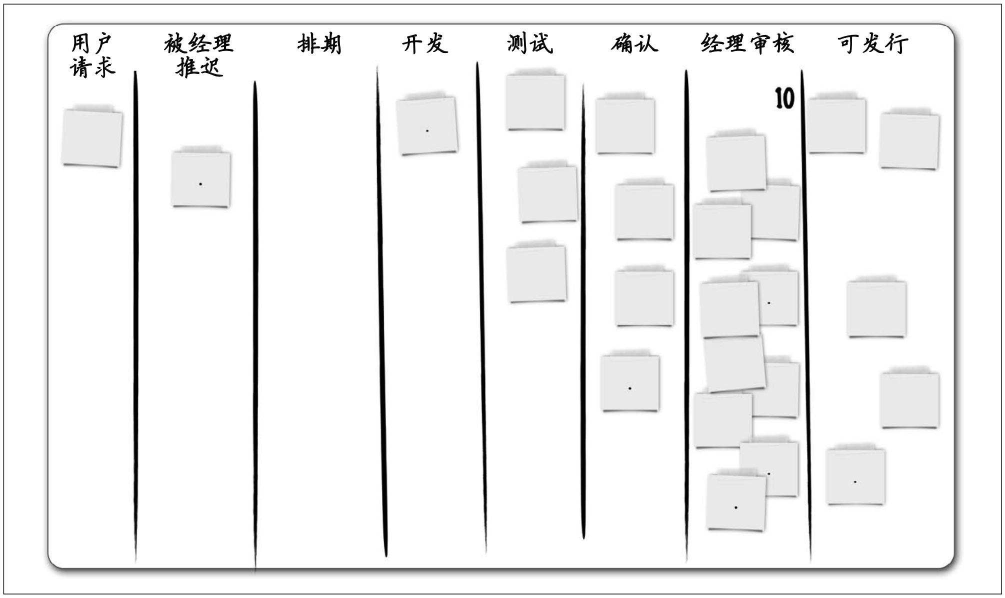


图 9-7：当看板的某一栏达到了它的上限，贴纸就不能再移入该栏，哪怕那些贴纸上的工作已经完成了也不行。开发团队会转而去做那些尚未达到上限的栏目中的工作

如果经理信守诺言并召开审核会议解除当前的僵局，那么工作就可以继续。进行中工作上限让经理更早给出他们的反馈，而团队则能够根据需要马上做出调整。如果发现有些特性可能需要推迟，经理就能够在项目早期就知道，而不必等到临近结束时才知道。这样一来，团队的麻烦就少多了，因为它不需要再把那么多已经完成的工作再扔进一个“已推迟”的单子里了。它正在做的工作可以马上产生价值，而不是在“已推迟”状态里待着，到最后对公司的价值也没那么大了（或者到后来用户因等不及而转向了竞争对手的产品）。

这种做法之所以有效，原因在于经理审核与团队调整的这个循环构成了一个反馈循环。当这个回路过长（比如，回路的长度就是发布周期的长度）时，反馈给项目的信息就变得具有破坏性了，而不是建设性的，因为它导致团队抛弃它已经做完了的很多工作。也就是说它导致了浪费。

增加一个进行中工作上限来解决超负荷问题可以把工作项在系统中存在的时间大大缩短。现在经理会在项目的整个周期里召开多次审核会议，这使得团队可以更早地进行调整，同时也让经理利用这些信息来在项目尽量早期的时候调整优先级，从而让有价值的工作不至于被浪费掉。

更好的是，团队与经理现在可以控制反馈循环的长度。比如，如果经理发现这样做很有用，可以批准把进行中工作上限降低到六个特性，这会导致经理的会议更加频繁，他们给团队的反馈也会更早。

为什么不把上限设为1，对每个特性都开个会？反馈循环不是越短越好吗？

不是的，如果团队花在审核会议和处理反馈上的时间比它实际做工作的时间还长的话，反

馈循环就不是越短越好了。当一个系统的反馈循环太短时，这个系统就会进入一种叫作信息过载（thrashing）的状态。这个时候有太多的信息被反馈给系统，而团队还没有足够的时间来对这些信息作出响应，新的一波信息又进来了。

团队应该避免把相同的工作项多次送进反馈循环中，因为这可能会阻塞整个系统。看板在这里又再一次发挥其作用了，它能够在这种情况发生的时候让我们注意到。比如，在上面例子中的看板上，当经理提出了反馈并需要重做某个特性时，他们会把它送回到积压工作表中。会有人把该特性对应的贴纸揭下来并移动到较早的一列。团队可以通过给这些贴纸加一个圆点或者什么其他的标记，来表示它们被往回移动过。这样就很清楚，这个特性肯定会重复进入反馈循环。当该特性再次进入工作流，它还会再次进入"经理审核"一栏，并占用一个进行中工作的名额。这可能会阻塞反馈循环。

要避免这个问题，团队可以从工作流程中去掉额外的回路，让流程对大部分特性变得更加线性。如果团队能够说服经理同意只进行一次特性审核，而且仅把一部分特性打回到积压工作表里，会怎样呢？如果经理能够信任团队，在大多数特性上会接受团队的意见，并不再要求在发布前再次审核这些特性，那么开发团队就可以在审核步骤之后再增加一组额外的开发和测试环节。

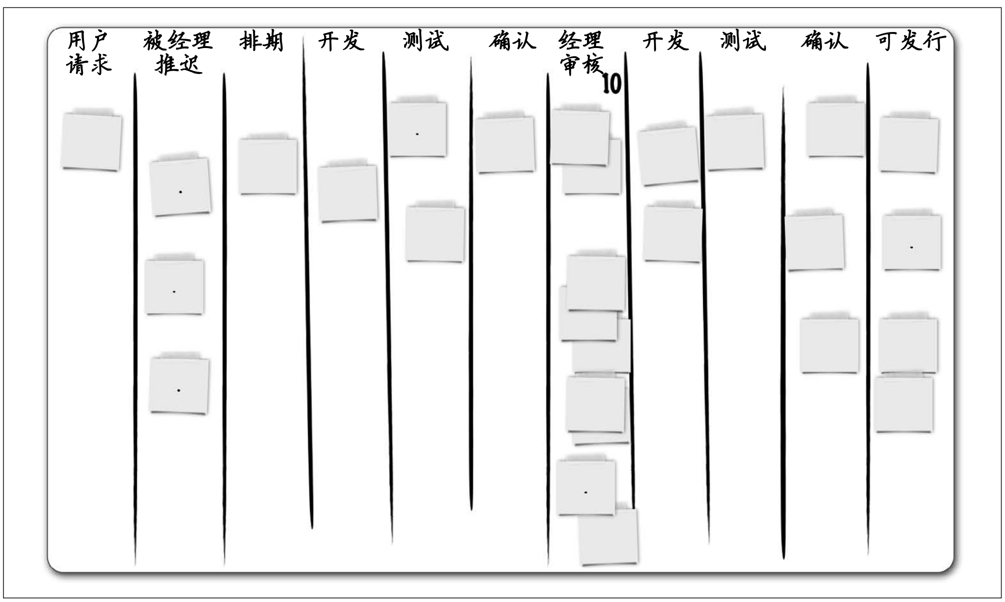


图 9-8：给看板增加更多栏目（避免过多的贴纸被打回前面的栏目）给了团队对流程更多的控制

这个工作流程在看板上看起来更长了。不过，因为我们使用了交付时间这个客观指标，并且通过实验调整了进行中工作上限，最终形成了现在这个工作流程，我们能够自信地说整个流程实际上更快了。而且我们可以继续测量交付时间和可视化工作流，以此来向团队和经理证明：尽管流程中的步骤增加了，每个特性通过整个流程并最终进入发行版本的时间其实变短了。不过，尽管那些客观指标对于持续改进项目和让经理相信项目正在改善很有用，但是你很可能不需要这些客观指标就能够得到开发团队的信服。团队所取得的结果本

身就会说话，因为在去除了不均衡和超负荷之后，项目的进展感觉上更快了。

要点回顾

- 看板方法是一种过程改善方法，或者说是一种基于精益思维帮助团队改进其开发软件和协同工作的方式的一种方法。
- 看板团队以现在的做法为起点，看到当下的全局，并追求增量式的、演进的改变来逐渐地改进整个系统。
- 看板方法在协作中提高，在实验中演进这一实践指的是用指标测量，做渐进式的改进，并且使用量化指标来确认改进确实是有效的。
- 每个团队都有一个开发软件的系统（无论它自己是否意识到了），而精益的系统思维正是要帮助团队理解该系统。
- 所谓看板是指用一个白板来将看板团队的工作流程可视化。
- 看板有很多栏目，每一个代表工作流程中的一个容器；栏目中的贴纸表示处于工作流程中的工作项。
- 看板上的是工作项,而不是具体任务,因为看板方法并非一个项目管理系统。
- 当看板团队**限制进行中的工作**时，它给看板的一栏增加一个数字，表示工作流中的该容器所允许存在的工作项的最大数量。
- 看板团队与其用户、经理以及其他利益干系人协同工作，来保证所有人都同意当一个栏目达到了它的上限时，团队将把注意力转向项目的其他部分，且不再将更多的工作项推进到已达上限的工作流容器。

9.4　测量并管理流量

随着团队持续地交付工作，它不断找出工作流程中的问题并调整进行中工作限制，从而让反馈循环能够提供足够的信息，又不至于导致信息过载。所谓系统的流量指的是工作项通过该系统的速度。当团队找到了一个交付的最佳节奏，并与合适的反馈信息结合起来，它就做到了流量最大化。去除不均衡和超负荷问题，并让你的团队一个任务一个任务地完成，会提升你项目的流量。当系统中的不均衡问题导致工作堆积时，就会导致中断工作并降低流量。看板团队使用管理流量（manage flow）这一实践，对流量进行量化，并积极采取措施来不断提高流量。

你已经知道当工作真正“流动”起来时是什么感觉。你会感觉你完成了很多东西，而且你没有浪费时间或者干坐着等其他人做一件事。处在一个流量很大的团队中的感觉是，每一天你都感觉自己在做着有价值的事情。这是每个人都追求的东西。

你也知道当工作没有“流动”起来是什么感觉。感觉就像你陷入了泥潭，勉强取得一些进展。好像你总是在等别人完成你需要的东西，或者等着别人做一些会影响到你工作的决定，或者等别人批准某个票证，或者不知道什么东西总是妨碍你的工作，即使你知道没有人有意要这么做。那种感觉是迷茫和脱节的，而你花费大量的时间给别人解释你为什么总是处于等待中。并不是因为分给你的工作量不足；你的项目团队很可能被安排了 100% 的

工作量，甚至可能更多。但是，尽管你的项目计划上说你们已经完成了 90%，感觉却好像还有 90% 的工作没有做完。

你的用户也很清楚当工作没有“流动”起来时是什么感觉，因为交付时间不断增加。似乎团队花费越来越长的时间才能响应用户的请求，而且即使是简单的功能似乎也总是做不完。

看板方法的核心就在于提升流量，并让整个团队都参与到提升流量的过程中。当流量提升时，因不均衡和交付时间长而造成的挫败感就会下降。

9.4.1 用CFD和进行中工作面积图测量并管理流量

看板是管理流量的一个重要工具，因为他能够将问题的源头可视化，并让你在最有效的地方限制进行中的工作。当你寻找堆积起来的工作并通过增加一个进行中工作上限来消除它时，你就是在为提升流量采取措施。进行中工作限制之所以有效，是因为你是在帮助团队把精力集中到那些妨碍工作流动起来的部分上。事实上，进行中工作上限的全部作用就在于此，它改变了团队当前的工作重心，并使得它去处理那些能够让流量均衡起来的工作，从而在不均衡问题显现之前就消灭它。

不过你怎么知道通过添加进行中工作上限你就真的提升了流量呢？我们再次回到精益思维上，精益思维告诉我们要使用量化指标进行测量，而测量流量的一个有效工具就是*累积流量图*（Cumulative Flow Diagram，CFD）。CFD 与工作进度面积图类似，但有一个区别：工作项不会从图中流出，而是在最后一个步骤所对应的带状区域中累积起来。第 8 章中你所见到的工作进度面积图中的各个色带对应的是价值流示意图中的各个状态，但本章的 CFD（以及工作进度面积图）中的每个色带都对应看板上的一列。

本章的 CFD 中还有一些额外的线，用来表示*平均到达速度*（arrival rate，每天新增的工作项的数目）和*平均工作存量*（inventory，工作流中工作项的总数）。CFD 中还可以展示平均*交付时间*（lead time，每个工作项在系统中存在的时间，就像我们在第 8 章中介绍的那样）。并不是所有的 CFD 都有这些线；但是这些指标对理解流经系统的流量都很有帮助。

使用 CFD 管理系统流量的关键在于寻找那些能够表明存在问题的特征。看板可以告诉你今天你的工作流中哪里存在不均衡，哪里是回路，哪里有其他方面的问题，并且帮助你通过添加进行中工作上限的方法来管理每一天的工作流。而 CFD 的作用则是让你能够看到“在一个时间段内，你的整个流程的表现如何”，这样你就可以采取措施来找出并修复那些长期的问题。

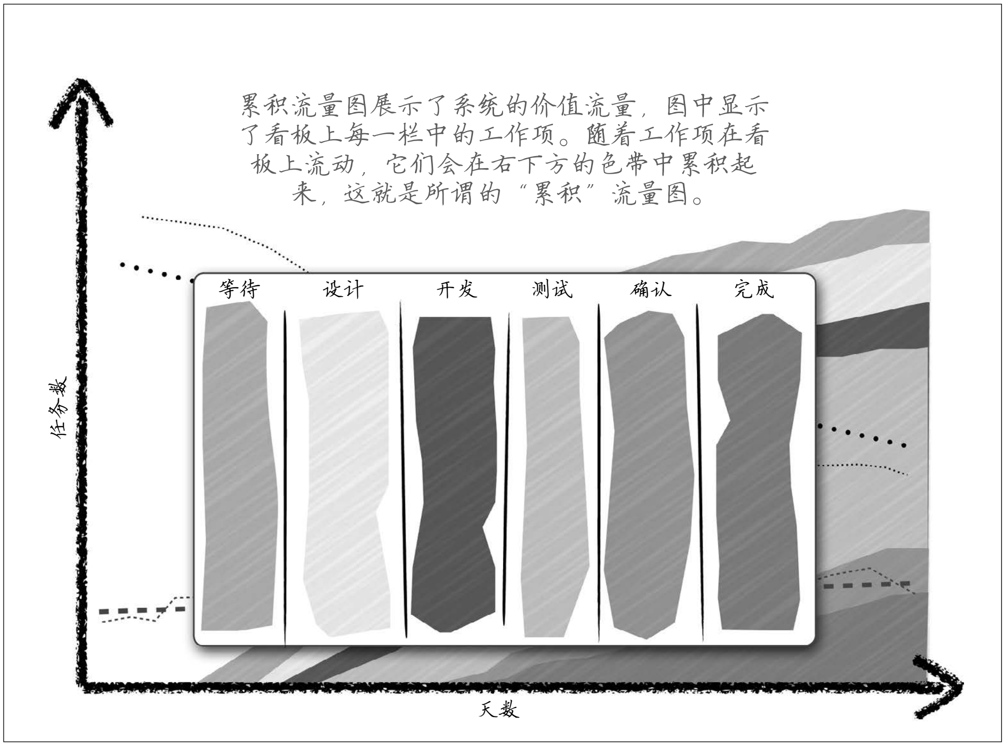


图 9-9：看板团队使用累积流量图，图中色带对应看板上的列。CFD 与工作进度面积图类似，不同之处在于前者的工作项不会从图中移除，而是累积起来，这样所有的色带都会随着时间的推移变得越来越宽

1. 绘制累积流量图并计算平均交付时间

要绘制一个 CFD，首先从一个工作进度面积图开始。不过，不要从价值流示意图中采集信息，你应该从看板上收集每个栏中的工作项数目数据。

接下来，你每天都要向图中添加两部分数据：到达速度和工作存量。要得到每天的到达速度，查一查看板第一列新增了几个工作项就行了。要得到每天的工作存量，查一查看板上的工作项总数即可。每天在 CFD 上给到达速度和工作存量添加一个点，把这些点连起来就形成了两个叠加在工作进度面积图上的折线图了。

大多数使用 CFD 的团队并不会真的在墙上一点点地画这个图；它会使用 Excel 或者其他支持生成图表的电子表格程序。除了管理数据比较方便之外，还有一个原因就是电子表格可以自动生成折线图。这些趋势曲线很有用，因为它们能够告诉你系统是不是稳定。如果这些曲线很平很直，那么系统就是稳定的。如果其中一个倾斜了，那么那条曲线所对应的指标就在随着时间发生变化。这时候你就需要给系统增加进行中工作上限来使系统稳定下来，曲线变平直的时候，你就知道系统稳定了。

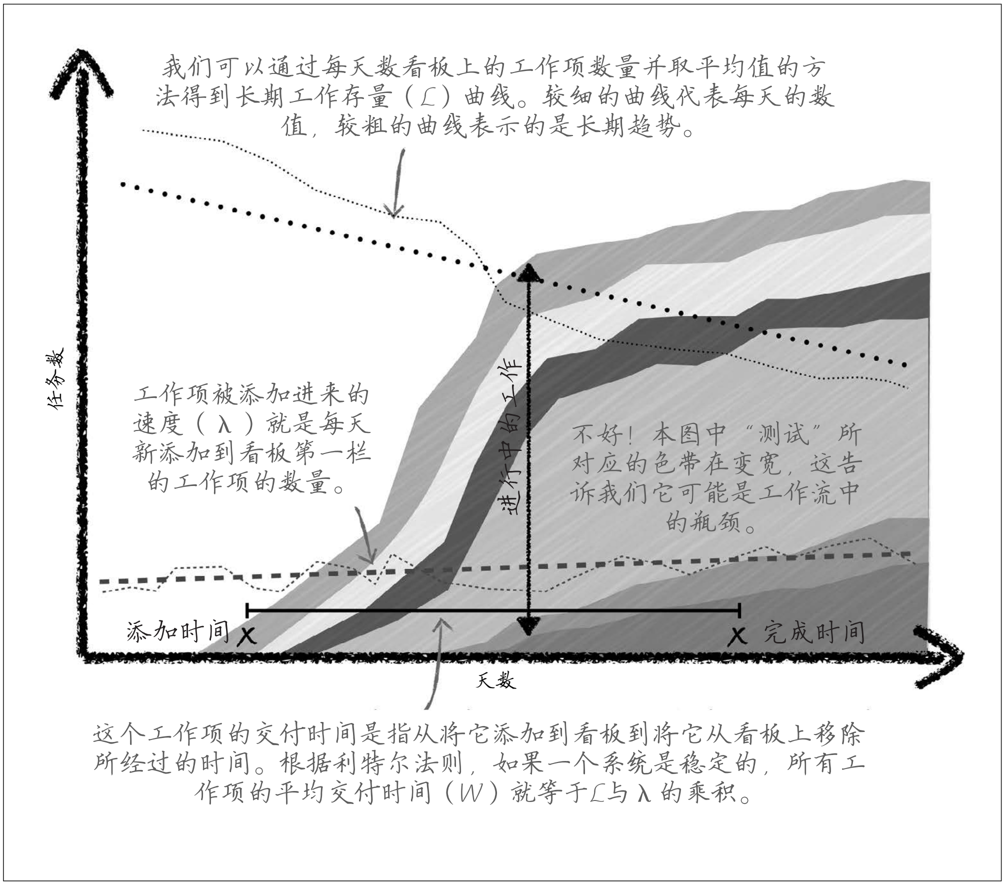


图 9-10：这是 CFD 中包含到达速度和工作存量的一个例子。任意给定时间正在进行中的工作量可以通过测量图的顶部与“已完成”色块间的差值得到。水平的黑色线段显示了系统中一个具体工作项的交付时间。我们现在暂时还没法计算平均交付时间，因为系统还没有稳定下来，长期的工作存量和到达速度曲线都存在一定的倾斜，说明它们不是常量

如果你看一下图 9-10 中的标注，你会发现我们把图中的那些量用字母表示了：用 L 来表示平均长期工作存量，用 λ（希腊字母 lambda）来表示平均到达速度（每天新增的工作项的数量），用 W 表示平均交付时间（用户等待团队完成一个请求的平均等待时间）。

请看 CFD 中的工作存量曲线，即顶部较粗的那条虚线。它的整体趋势是下行的，也就是说整体工作存量随着时间推移而下降。这表明很多工作项流出了系统（被完成了）却没有足够多的新工作项进来替代它们。可是，如果你看一下图底部的到达速度曲线，会发现到达速度实际上是在增加中的。

如果我们持续跟踪这个项目，会发生什么呢？工作存量会再次增长起来吗？会不会某一次发布把系统中的工作项都清空呢？如果到达的特性比完成的特性多，那么长期来讲工作存量会逐步增加，而团队会感觉到这种趋势。团队会慢慢地有更多的工作要做，并会感觉到

它的时间不够用。这是系统没有流动起来时的那种深陷泥沼的感觉。

幸运的是，我们知道如何克服这一问题：添加进行中工作上限。团队可以同时实验和反馈循环来找到适合其系统的上限值，而如果它把这个值设置对了的话，那么最终工作项的到达速度会与团队完成它们的速度达到平衡。长期工作存量的趋势就会是水平的，长期到达速度的趋势也是一样。一旦达到这样的状态，系统就*稳定*了。

而当一个系统稳定下来的时候，上述各个数量之间存在一个简单的关系，这称作*利特尔法则*（Little's Law），该法则是队列理论的一部分，它是以它的发现者 John Little 命名的，John Little 在 20 世纪 50 年代提出了该理论，并被很多人认为是市场营销科学的先驱。虽说这个法则中有一个希腊字母，但是你不需要太多数学知识就能使用它。

$$L = W \times \lambda$$

说白了，意思就是如果你有一个稳定的工作流程，那么平均工作存量总是等于平均到达速度乘以平均交付时间。这是一个数学法则：它是已经过证明的，而且如果一个系统是稳定的，这个法则一定成立。反过来也一样。

$$W = L \div \lambda$$

如果你知道平均工作存量和平均到达速度，你可以算出平均交付时间。事实上，计算平均工作存量和平均到达速度非常简单：每天把你看板上的工作项总数和新增到第一栏的工作项数目记下来就行了。我们在上面的 CFD 中用较细的虚线来表示这二者。如果你的系统是稳定的，那么过一段时间你就可以算出平均工作存量和平均到达速度。用平均工作存量除以平均到达速度，你就得到了平均交付时间。

现在停下来想一想。交付时间是量化用户满意度的最好方式之一：交付得快，用户就高兴；拖好长时间才交付，用户就越来越不满。过长的交付时间是质量问题的很好指示器，正如 David Anderson 在他的《看板方法》一书中所指出的那样。

> *更长的交付时间好像总是与显著较差的质量有关系。事实上，大约 6.5 倍的平均交付时间增长会导致初始缺陷增加超过 30 倍。更长的平均交付时间是由更大量的进行中工作导致的。所以，管理层赖以改进产品质量的方法就是减少进行中工作的数量。*

你的交付时间完全是由工作项进入系统的速度和它们流出系统的速度决定的，而进行中工作上限让你能够控制流动的速度。

那么这一切意味着什么呢？这意味着当你的系统稳定的时候，你可以仅仅通过“不开始做新的功能”就能减少客户的等待时间。

2. 使用CFD实验进行中工作上限的值并管理系统流量

看板方法的一个核心思想是：一旦你把工作流程可视化了，就可以测量流量，让系统稳定下来，并确实通过管理你开始新工作项的速度来控制项目的交付时间。

这可能听起来比较抽象。设想一下你如何把 CFD 用到我们大家都很熟悉的简单系统上也许会有帮助：医生的诊所。假设为了做一系列的检查并与医生讨论检查结果，你不得不拜访某个医生好几次。你发现如果你和医生约在诊所刚开门不久的早上，你就不需要等太长时间。但是如果你约在了较晚的时间，你就得在候诊室等上好一阵，预约的时间越晚，

等待的时间也越长。很显然，这个系统是不稳定的。你如何使用一个 CFD 来使其变稳定呢？

第一步是将工作流程可视化。假设每个患者从在候诊室坐下来开始。后面，护士会把患者叫到一个化验室，在那里对患者称重、量血压、测体温。然后患者就等着见医生。在这个诊所，一共有 5 个化验室和 2 个医生，这些化验室和医生都一直被占用着。更重要的是，这意味着同时在化验室中的患者不会超过 5 个，看医生的患者不会超过 2 个。这些就是进行中工作上限，是由现实系统的约束决定了的。

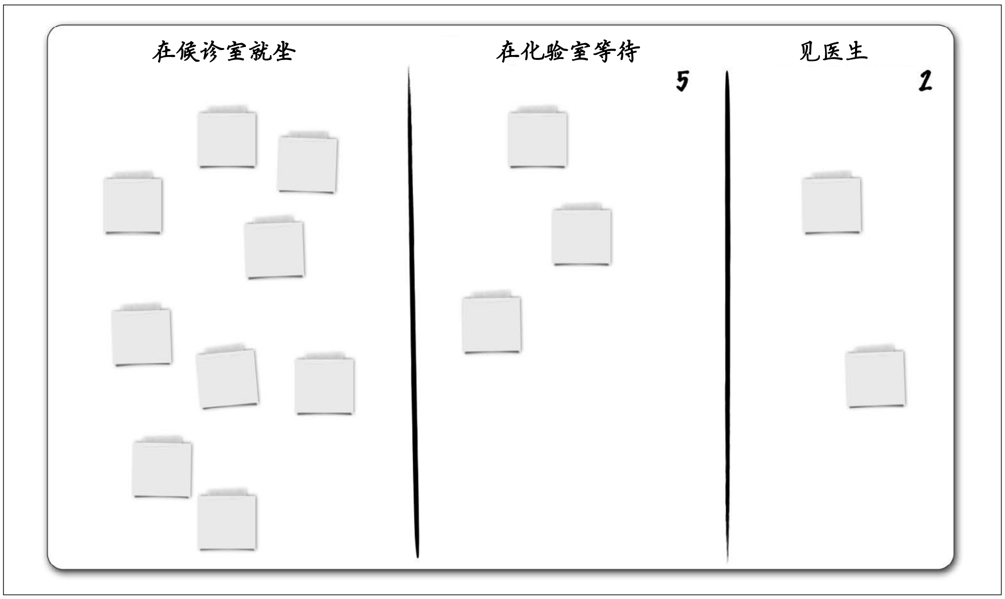


图 9-11：一个诊所的看板。第一栏表示的是正在候诊室中的患者，第二栏表示化验室中的患者，而第三栏表示正在见医生的患者。共有 5 个化验室和 2 个医生，这些是自然形成的进行中工作上限，所以也画在了看板上

医生反感那些预约时间较晚的患者抱怨等待时间长。更糟糕的是，医生在每天晚些时候都会感觉到看患者的时间很赶，他们担心自己可能无法作出最佳的医疗决策，因为他们处在让患者尽快完成就诊流程的压力之下。看板方法能帮助这些医生减少等待时间并提供更好的医疗服务吗？

我们来试试。首先我们给该诊所典型的一天绘制一张 CFD。我们让诊所的工作人员记录每个 15 分钟内进入诊所就诊的患者数量。这就是到达速度，即字面意义上的到达诊所的患者数量。然后我们可以数一下候诊室和 5 个化验室中患者的总数来得到每个 15 分钟区间里的工作存量。每当有人到达诊所，看板的第一栏就增加一张贴纸。当患者从候诊室到了化验室时，该贴纸就移动到第二栏。当医生开始看患者的时候，贴纸就移动到第三栏。医生看完患者之后，贴纸就从看板上拿掉。诊所的工作人员可以对每个 15 分钟区间内各栏的贴纸书进行计数并记录下来。

现在诊所就拥有了绘制 CFD 所需的全部数据。

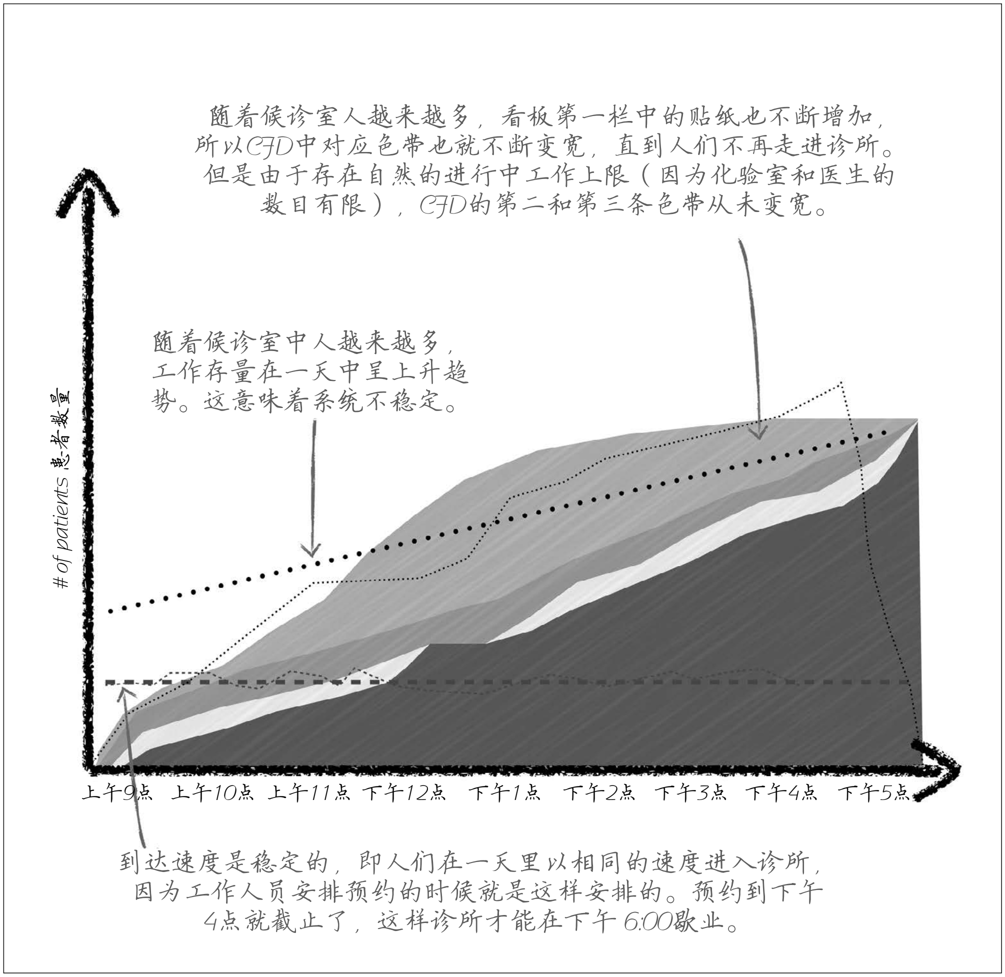


图 9-12：这张 CFD 显示了诊所的患者流量。候诊室在一天当中越来越满。你能猜出为什么第三栏对应的色带在中午 12:15 到下午 1:00 之间消失了吗

这个系统还没不稳定。到达速度是稳定的，因为诊所的工作人员安排的患者预约就是让他们以固定的速度到达。诊所不想加班到很晚，所以下午 4 点以后就不安排预约了。有些人确实有来晚的时候，但是到达速度的整体趋势是平缓不变的，所以一天中到达速度是基本不变的。

但是，工作存量的趋势可就不是平缓的了。它整体呈一个向上的趋势，因为工作存量不断地增加。这很合理；候诊室中的患者数量在一天中也是不断增长。那么诊所的工作人员如何利用这些新的信息来改善患者服务并削减等候时间呢？

工作人员要做的第一件事就是让系统稳定下来，而我们为此可用的工具是设置一个进行中

工作上限。工作人员将使用看板方法中的“在协作中提高，在实验中演进”这一实践来一起确定具体的上限值，同时以这个值作为起点。通过查看采集到的数据，大家决定给候诊室设置 6 个患者的上限。但是需要做一个棘手的决定：医生必须得同意，如果候诊室已经有 6 个患者在等候了，那么工作人员必须给约在接下来一小时的较低优先级患者打电话，把他们的预约延后（但是工作人员会想办法处理重症患者，从而不影响患者服务）。工作人员还会问候诊室中的患者是否有人愿意自愿推迟他们的预约，并保证新的预约将享受较高的优先级。这是一种需要明确的说明的新规则。工作人员通过给看板增加一个进行中工作上限来做到这一点，同时，还在门口的桌子上贴了一张大布告，告诉患者以后将采用这种新的规则。

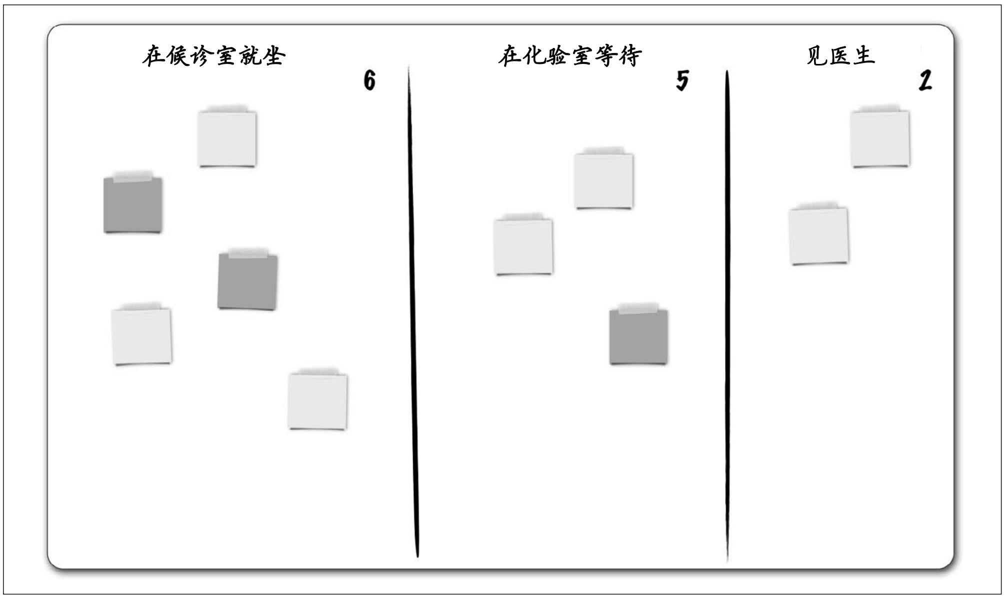


图 9-13：工作人员设定了新规则，规定候诊室中最多只能有 6 个患者。工作人员在候诊室人数达到 6 人时给后续患者打电话重新安排预约，从而保证规则得以实施。在看板上，工作人员用粉色贴纸来表示那些不能调整预约时间的重症患者

这需要练习一段时间，不过，过了些日子诊所的工作人员就习惯了新的规则。他们发现他们需要把患者的抱怨也纳入考虑。他们决定通过定义服务等级来做到这一点：他们使用粉色贴纸来表示无法调整预约时间的重症患者，并继续使用黄色贴纸表示普通患者。这使他们能够为那些最需要的患者提供及时的服务。

结果是这么做起作用了！诊所的工作人员发现一旦他们设置了进行中工作上限，就不必把预约截止到下午 4：00 也可以在 6：00 之前结束营业，他们可以给患者安排到最晚下午 5：40，只要在 4：40 以后不安排重症患者即可（他们对这一点也作出了明文规定）。显然，如果有重症患者来就诊，医生肯定会接诊该患者（或者把他送进急诊室），但这并不常见，而且由于诊所工作人员聪明且有责任感，他们能够视情况做出适当的处理。患者更加满意了，因为不用像以前那样等待那么长时间了。

9.4.2 用利特尔法则控制系统的流量

诊所工作人员非常严肃地看待看板方法的“在实验中改进”这一实践。而通过实验，他们发现了很有趣的现象：一旦他们找到一个合适的进行中工作上限，就能控制患者的等候时间。如果他们每小时安排更多的预约，候诊室中就会有 4 到 5 个患者，而且这些患者需要等候更长的时间；而如果他们每小时安排的预约少一些，等着见医生的患者就只有 2 到 3 个，而且这些患者的等候时间也会更短。这让工作人员感觉到，他们第一次真正能够控制这个过去曾让他们很头疼的系统了。

那么到底发生了什么呢？

诊所工作人员发现的是：在一个稳定系统中，工作存量、交付时间和到达速度之间存在着一定的关系。比如，如果工作人员每小时安排了 11 个患者到达诊所（到达速度 λ 是 11/ 小时），同时诊所一天中的平均工作存量是 7 个患者（工作存量 L 是 7），那么根据利特尔法则，患者的平均等候时间如下所示。

$W = L \div \lambda = 7$ 个患者 $\div$ (11 个患者 / 小时) = 0.63 小时 = 37 分钟

但是，倘若经过一些实验，他们发现每小时安排 10 个患者到达能够把平均工作存量降低到 4 个患者，那将会怎么样？一天中会有一些化验室都被占满的高峰时段，不过大部分时间候诊室有一个患者，化验室有一个患者等着见医生，还有两个患者在化验室跟医生交谈，这种情况下患者的平均等候时间如下所示。

$W = 4$ 个患者 $\div$（10 个患者 / 小时）= 0.4 小时 = 24 分钟

通过使用看板和 CFD，同时通过实验调整进行中工作上限，诊所的工作人员发现他们可以仅仅通过每小时少安排一个患者，就把患者的平均等候时间缩短将近 15 分钟。

这种方法之所以有效，是因为利特尔法则告诉我们，一个稳定系统中，交付时间仅受到两样东西的影响，即工作存量和到达速度，而进行中工作上限可以让你控制其中的一个。当你在看板上增加进行中工作上限时，你就能够减少不均衡，于是让工作存量不至于积累起来。这就给了你一个简单直接的缩短交付时间的方法：降低到达速度（比如，在团队有时间处理一个工作之前，把它放在一个积压工作表里面，就像 Scrum 团队所做的那样，或者减少每小时安排的患者数量）。

因此，诊所的工作人员可以使用利特尔法则来计算平均交付时间。但是就算他们从来没有去计算这个时间，也依然会受到它的影响。原因是利特尔法则对稳定系统一直适用，不管团队是否意识到它的存在。这会影响到你的项目，因为它不仅仅是理论。它是一条被证明了的数学法则，适用于任何具有稳定长期工作存量的系统。

现在我们看过了一个简单的例子，让我们再来看一个更加接近现实生活的例子。假如每三个星期，你的整个团队就需要投入到支持生产环境中的发行版本的工作中。

不幸的是，对团队来说，情况并不是太好。最开始都很顺利，可是随着时间的推移，问题开始显现。尽管所有的支持工作最后都完成了，但是开发团队感觉没有足够的时间。所有人都感觉这个月总是比上一个月压力更大，大家也都知道如果他们不把支持工作尽快完成，下个月的状况将更加糟糕。

这种感觉对很多团队来说都不陌生。感觉像是项目在慢慢地陷入流沙，如果继续下去，最后情况会坏到团队中的每个人感觉根本没有时间来思考他们的工作。我们在第 7 章中学习过这种环境对于代码造成的破坏：开发人员会开发出设计不良的软件，而且会造成一个技术债务累积、难以维护的代码库。

如何解决这个问题呢？我们无法简单地通过每天盯着看板看就得出答案，因为大部分时间看板看起来多半是健康的。当开发团队在处理支持发行版本的工作时，看板上有一些额外的贴纸，不过这都是意料中的。最终这些贴纸会从看板上移除，而一切看起来都跟往常一样。但是团队的感觉可跟往常不同（或者更糟，感觉确实正常，但绝非感觉好！）。

我们学习过的工具能够帮助找出并解决这个团队面临的问题吗？我们一起来看一下。

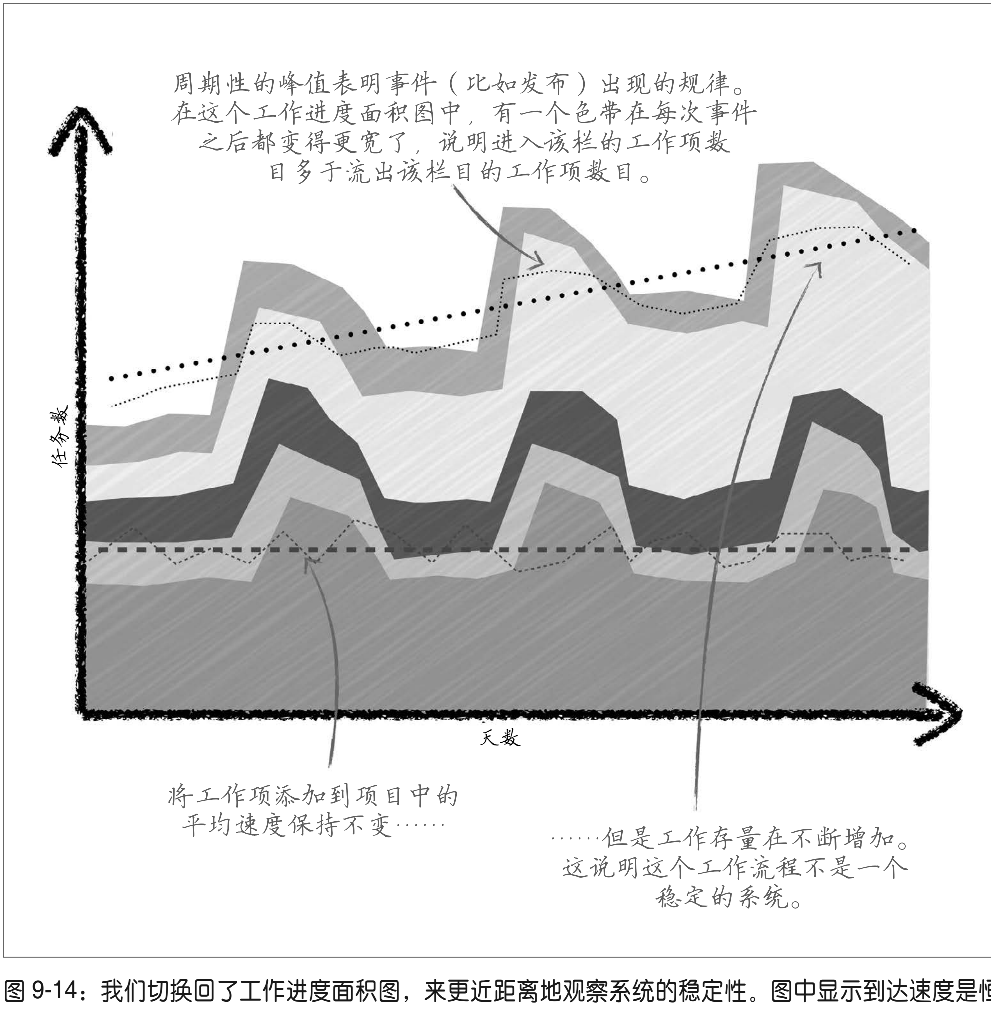


图 9-14：我们切换回了工作进度面积图，来更近距离地观察系统的稳定性。图中显示到达速度是恒定的，但是平均工作存量随时间逐步在增加，也就是说系统是不稳定的

每次生产发布的额外工作导致 CFD 中出现一个明显的峰值。记住，每个色带对应看板上的一栏，而色块垂直方向上的宽度代表当天该栏目中的贴纸数量。在这张 CFD 中，有一个色带在每次发布之后都变得更宽，同时这也导致整个图的高度随着时间的推移缓慢增加。

工作进度面积图让问题的原因变得明显了：工作在该栏形成了堆积，因为随着时间推移，进入该栏的工作比流出该栏的多。如果团队不做出改变的话，每次新发布都会让它更加落后，而那条色带会在每次发布后不断变宽。同时我们看到长期工作存量在上升，说明我们的系统不稳定，没有进入流动状态。难怪团队成员会感觉他们在陷入流沙。

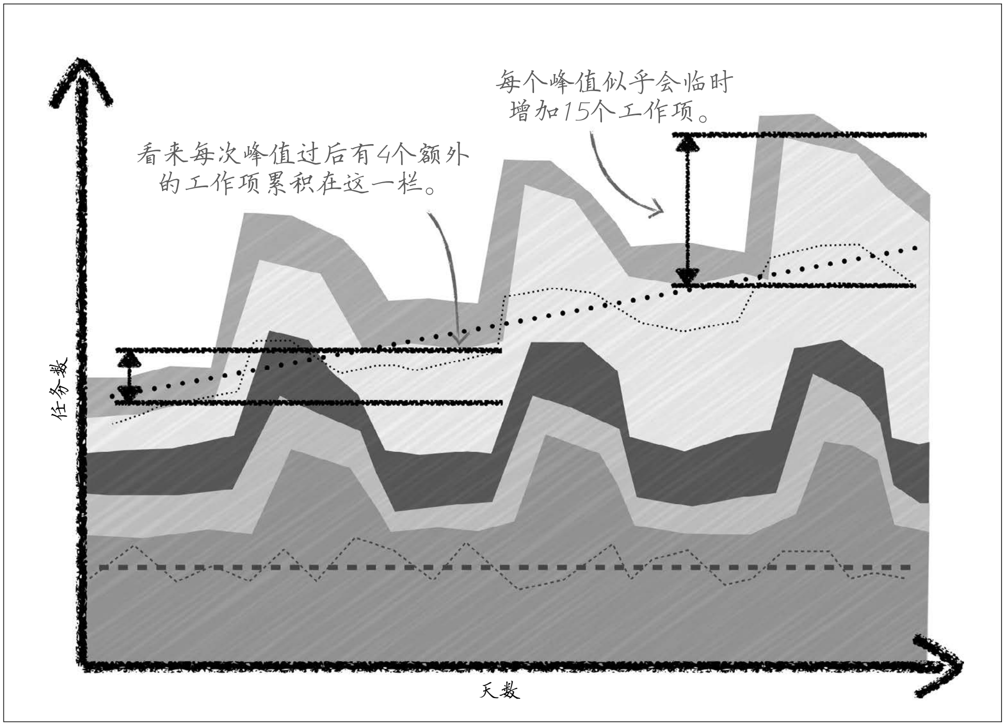


图 9-15：这张工作进度面积图同时也包含一些线索，可以帮助我们找出如何把系统稳定下来的方法。平均工作存量在每个周期性的工作峰值后都会增加。如果我们能够知道有多少额外的工作存量发生了累积，就可以帮助我们选择一个实验工作上限的好出发点

团队可能没有意识到工作在看板的某一栏“卡住”了。工作进度面积图让这个问题更容易发现了，因为该栏对应的色带会越来越宽。幸运的是，我们已经有办法去除这种不均衡了：增加进行中工作上限。而测量有多少工作项“卡在”了那一栏中可以帮助我们给工作上限找一个好的初始值。

周期性出现的峰值依然存在，但是总的任务数量不再增加了。通过增加一个队列，我们阻止了额外的工作流入系统，进而增加了整个项目的总体流量。当超负荷的那一栏达到了它的上限，团队就把注意力转向更早的栏目中的工作项。你能够在工作进度面积图中看到这一点：较底部的色带的凸起比未设上限前变小了。

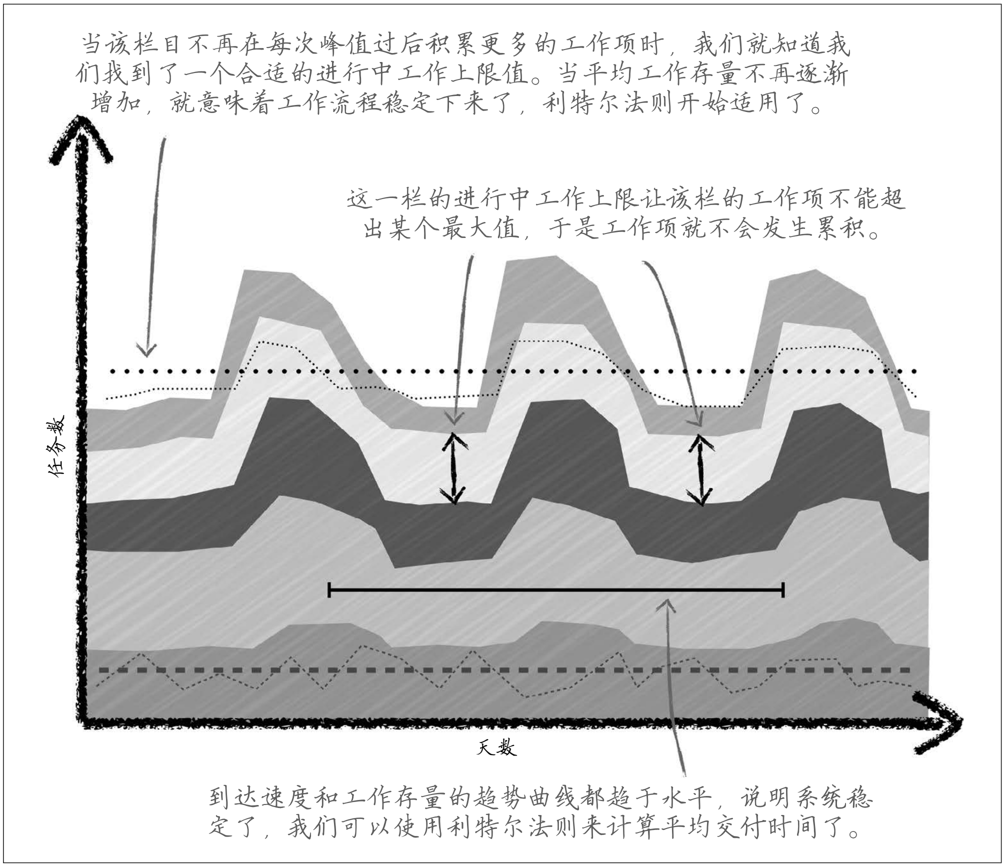


图 9-16：当你找到一个合适的工作上限值时，看板的对应栏就不再累积工作项了，面积图中的色带也就不会越来越宽。这在 CFD 上看起来是什么样子呢？你觉得在将这个问题可视化这一点上，CFD 会比工作进度面积图更合适吗

不过，看板团队绝不是设置一个进行中工作上限之后就完事了。它会实现一个反馈循环：随着项目的继续推进，它会不断根据新的信息调整该上限的值。如果累积依然存在，它就尝试不同的上限值，直到找到一个能够实现最大流量的值。看板团队非常重视实验：它不会随机地设置上限值；它会先提出一个关于上限会如何影响系统的假说，然后进行实验，并通过采集量化指标的方式来对假说进行验证。这就是团队如何做到“在协作中改善，在实验中演进”的方法。

随着团队互相协作，演进其系统，它会感觉到日常工作中的流量增加了。支持任务不会让它感到它推迟了重要的开发工作，因为它把支持工作也看作是需要开发的功能。通过把支持工作的工作项放到看板上，团队实际上是把支持任务作为项目的一等公民看待，从而能够专注于它们并更好地完成它们（而不是把它们塞进工作日程里并草草赶工，与此同时也给它自己制造了更多的麻烦）。

这同时还带来一个额外的长期好处。很多的支持问题是由团队因赶工而欠下的技术债务导

致的，现在团队有足够的时间把事情做好，它可能会发现将来的支持任务变少了。同时，团队变得更加专注，还能享受一个更加让人精力充沛的工作环境，这也让它能够开发出更好的软件。

如果团队还是得做所有的生产支持工作，那么原本在做的那些工作怎么办呢？这些工作难道不会在堆积起来吗？

不会的。那些额外工作不会在工作流程中堆积的原因是，它们一开始就没有被添加到工作流程中。团队原先之所以会感觉到有压力，是因为它的上级要求它把全部精力放在开发软件上，但是与此同时，还要求它每隔几周就停下手头工作转而专注于支持任务，又不能影响开发工作。这是它的上司的选择。或者更准确地说，这是它的上司不愿意作出的选择。该上司的神奇思维让他认为团队能够处理所有的开发工作，同时还能每隔几周处理额外的支持任务。看板方法中的队列将强迫该上司选择团队应该处理哪些工作。

稍等一下，如果支持工作占满了队列呢？还怎么处理开发工作？

如果支持任务把队列塞满了，那说明团队的上司选择了在支持和开发工作之间，优先处理支持工作。后者就是团队的最重要任务，不管上司嘴上是否承认这一点；把支持工作项放到看板上是一种让团队给予支持任务足够注意力的方法。如果团队无法继续做开发任务，那不再是团队的错。当然，与去做支持工作相比，很多开发人员更愿意做开发工作；他们可能不会喜欢变成一个全职支持团队。但是这比需要同时对全部的支持和开发工作负责要强得多。

团队的上司现在也有了更多关于团队工作进度的准确信息。这是第 3 章中讲过的敏捷软件开发的原则之一：可用的软件是进度的主要衡量指标。以前，超负荷的团队还想能够在开发软件的同时还做支持；不容易注意到的是，额外的工作量导致它开发出更难以维护的、质量不佳的软件，并且可能直接导致了部分支持问题。现在团队虽然交付更少的软件，但上司却得到了更准确的进度指标。这将使他更难以使用神奇思维去假装团队能够处理根本不可能做到的工作量。如果他想增加产出，他要么就提高开发工作的优先级，使之超过支持工作的优先级，要么就需要雇佣更多的人手。但是他不能轻易地把责任推给团队。

9.4.3 用进行中工作上限管理流量，自然地创造缓冲

开发人员需要日程安排里有一定的缓冲空间。他们需要这种缓冲来保证他们有时间高质量地完成工作。我们在极限编程的几个章节中看到，当一个开发人员感觉他没有足够的时间思考他的工作时，就会偷工减料，于是欠下技术债务。他的脑子里想的是尽快完成手头的工作，因为总是有更多的工作要做，而项目一直都是落后于计划的。

这种“一直落后，一直赶工”的气氛几乎没有给创造力留下任何空间，同时也扼杀了任何创新的可能。它同时也伤害了质量控制，因为总是感觉落后进度的团队常常会排除那些不直接生产出代码的工作。这就是极限编程团队把留出缓冲作为它的一项主要实践的原因。

看板团队一样看重缓冲，并且也理解缓冲对每个团队成员做到最好的能力的影响。这也是它限制进行中工作数量的一个主要原因。当团队使用看板方法来改进它的流程时，它会使用所谓的*交付节奏*，而不是严格的时间限制。为了做到这一点，它承诺以固定的周期交付软件（比如，一个团队可能会承诺每隔 6 个星期发布一次软件），但是它不会承诺哪些具

体的工作项会包含在一个发布版本中。它把交付哪些具体的工作项交给它信任的系统来决定。如果它去除了流程中的超负荷和不均衡，那么它会自然地得到一组可以包含在一次软件交付中的已完成工作项。

那么，到底是什么导致了程序员或者其他团队成员感觉没有足够的时间呢？导致这种感觉的原因是，你清楚地知道剩余的工作量，而你目前正在做的工作正阻碍着项目的其余部分。如果一个人觉得自己正在做的工作是妨碍其他工作推进的一个绊脚石，他就会尝试尽快完成它。

这就是为什么当团队提出的进行中工作限制得到经理的批准时，大家都有一种惊喜的反应，因为他们感觉得到了解脱。

在对进行中工作进行限制之前，额外的工作好像总是能溜进冲刺中，而团队不得不依赖缓冲空间来处理这些额外的工作。每次冲刺开始的时候它都需要假定它只有 70% 的工作量是确定下来的，因为急躁的上司和用户总是会有办法塞进最后关头的修改和紧急的请求来占满另外 30% 的工作量（或者更糟，40% 或更多！）。

设置了进行中工作上限之后，还有更重要的，即对遵守该上限达成共识之后，这些额外的请求还是会进来，但是现在团队不必试图去消化这些计划外的工作了。相反，新的工作会进入一个队列，该队列是因在工作流的某个地方增加的一个进行中工作上限而引入的。压力比以前小了，因为团队成员知道不会有无穷的工作堆积成山。他们已经把他们的工作流程管理起来了（可能是使用 CFD），所以他们知道队列的工作上限设置在了一个合适的值上，能够给予他们足够的时间。

这就是为什么很多看板团队最后在看板的每一栏中都设置了进行中工作上限。

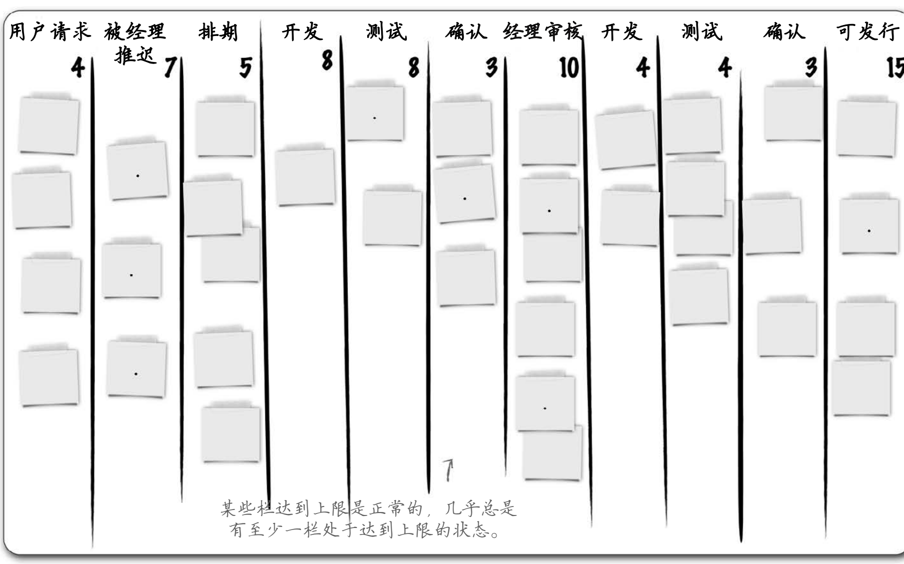


图 9-17：为看板的每一列设置工作上限可以帮助团队将整个项目的工作流量最大化

这可以帮助团队控制开发过程中每一个环节的流量。甚至连“可发行”一栏都设置了上限。如果已经有很多工作完成了，在等待发布，团队可以去做其他工作来为下一个发行版本做准备，同时它有了更明确的信息从而在将来调整它的交付节奏，减少发行版本之间的时间。

团队不可能在最一开始就让所有人就全部的工作上限达成一致；这也是为什么看板团队会遵循“在协作中改善和在实验中演进”的循环。在第一轮的改进之后，看到了工作上限能够帮助团队更快速地开发软件并缩短交付时间，经理就更有可能同意在接下来的改进周期中添加更多的工作上限。

控制整个项目的流量能够帮助团队中的每个人放松并专注于手头的工作，不必担心任务会在他们身后堆积成山。他们可以相信工作上限会把混乱排除在外。同时他们知道如果混乱开始出现了，也能够在对工作流程进行量化分析时发现之，而且可以调整工作上限和交付节奏来保持工作的专注和顺畅。

9.4.4 让过程策略明确统一

如果你让一个高效的 Scrum 团队中的每个人写出一个详细的文档，说明他们是如何编写软件的，会发生什么？很有可能几乎所有的说明都差不多是一致的。那是因为团队中的每个人都熟悉 Scrum 规则，而且他们都一直在按照这些规则进行工作。高效的 Scrum 团队中的每个人都对整个软件开发方法有着共同的认识，而且 Scrum 项目的规则也足够简单，每个人都能够理解。

另一方面，如果你让一个传统的、低效率的瀑布式团队做同样的事，会发生什么？很有可能团队成员的认识会是零散的（正如我们在第 2 章中看到的）。每个人可能会写下他们各自的日常工作是怎么做的：开发人员会写关于编码的内容，测试人员写的则是如何做测试，业务分析员写的则是如何收集需求，等等。项目经理可能有一个更全面的视角，因为她需要知道每个人都在做什么才能做出一个项目计划来，所以她也许能够写出一个包含了所有人工作的描述。但是她或许也只是描述了她自己如何计划和追踪项目的那些步骤。

有时候一个复杂的过程很重要也很有用；另外一些时候，它就是多余的、官僚的。比方说一个团队的习惯做法是，当规格文档发生变化时，会发生大量的电子邮件沟通，开发团队必须要等到有足够的人同意该项更改才能开始实施它。这是一个流程，即便它不是一个成文的规定，可它确实存在于团队的共识中。你怎么知道这个更新规格文档的过程是有用还是没用？答案就在看板方法的*让过程策略明确化*这一实践中；换句话说，把团队的工作方式写下来，并给每一个受其影响的人看。也许确实需要一个复杂的过程。比如项目经理也许会指出这个规格变更过程是监管部门的要求。但是更常见的情况是，单是把一个不成文的策略写下来就能让大家纷纷摇头，并同意这个流程是欠考虑的。

看板团队不需要编写很长的文档或者创建大型的 Wiki 来建立明确的规则。它的规则可以很简单，比如说就是看板上各栏中的工作上限。团队也可以把它关于“完成的定义”或者“退出条件”的要点写在每个看板栏目的底部，这样团队成员就很清楚什么时候可以把工作项推进到工作流的下一个容器中。当这些规则是通过协作和实验逐步建立起来的时候，这种做法尤其有效，因为这意味着每个人都清楚规则为什么要这么定。

复杂的流程和不成文的规矩常常会随着时间的推移而出现，这种情况在那些视角割裂的团队中尤其常见。一个复杂的变更控制流程可能会出现，因为业务分析员跟不上大量最后关头的变更，而且在软件没能满足用户需求的时候在整个团队面前被大声地批评。他可能就会增加一个复杂的过程来控制软件的范围，并且保证一切都落到纸面上，这样如果将来有谁改变主意了，他可以用白纸黑字来确保自己不受责备。我们很难对这个业务分析员为了自保而搞出官僚流程的做法加以责难；因为这也许是他对一个他应当负责的软件规格实现控制的唯一方法。

设置工作上限就是选定了一个规则，而这个规则有效的前提是：每个人都尊重一个共识，即当一个队列被占满、达到了它的上限时就不再把更多的工作添加到该队列上。把这个共识写下来，尤其如果是写在一个显眼的地方，比如一个看板上，这有助于确保每个人能够遵守这个共识。每当过度热心的经理或者用户尝试往队列里添加更多的工作时，团队可以指着写下来的规则拒绝他们的要求。当团队需要从队列里移除一个工作项以便给某个紧急的请求腾出空间时，写明了的规则也为之提供了更充分的理由。就此类问题跟经理或用户的沟通会更加顺畅，因为他们已经同意了一个白纸黑字写在那里的规则。

9.5 看板方法下自然发生的行为

> 塔金，你的手攥得越紧，就有越多的星系从你的手指间溜走。
>
> ——莱娅公主

回想本章前面的诊所的例子。如果你当初让一群上令下行式的项目经理来帮助诊所的工作人员来改进他们的流程，会发生什么情况呢？上令下行式的项目管理的第一步通常是估计，所以他们可能会让医生估计一下在每个患者身上花费多长时间。这需要医生、护士和工作人员在一开始做大量的分析，但是这可以让项目经理控制整个系统并为所有的资源（化验室、医生和护士）创建一个完整的日程表，进而对整个就诊流程进行全方位的微管理。

这将是一种僵化的工作方式；更重要的是，它将是非常没有效率的。医生的专业是看诊，而不是估计。项目经理拿出一个理想状态的日程当然是有可能的；但更有可能的是，他们会搞出一个不现实的或者没有效率的安排，因为医生给出的估计不够准确。这就是为什么看板方法把整个系统当作一个整体来看待。一个使用看板方法的团队不是去尝试对每一个微小的活动都进行微管理，而是使用系统思维去理解、量化、并渐进式地改进整个系统。这样的团队接受单个工作项之间可能会有所不同这一现实，但是当不均衡、超负荷和无价值的活动（Muda、Mura 和 Muri）被去除时，系统作为一个整体，其行为是可以预知的。

当一个团队使用看板方法来逐渐改进其开发软件的流程时，一个有趣的现象常常会出现：公司的其他部分也开始发生转变。考虑那个想要缩短交付时间的团队的例子。在该团队使用看板方法之前，经理和用户责备团队没能足够快速地响应用户的请求；团队的反应则是拿出项目计划来“证明”一切都是正常的。这最终导致冲突和负面情绪。

看板方法帮助解决了这背后的问题。团队用五个为什么技巧来找出了交付时间过长这个问题的根源，并引入了工作上限来帮助解决它。

可是，这种做法到底是怎么把问题解决掉的呢？让我们回到本章讨论的前两个看板方法实践（*在协作中改善，在实验中演进和实现反馈循环*）来理解一下到底发生了什么。看板方法做的第一件事就是改变所有人如何“看”（这里的看，就是字面意义上的“用眼睛看”）工作流程。用看板把工作流程可视化，这给了大家一个更全面的视角。经理会看到工作堆积成山，这有助于说服他们同意设置工作上限，并更经常地召开审核会议。这是工作上限如何导致行为变化的一个例子。

再比如，考虑一个团队不断收到来自好几个不同经理的要求，并需要在他们之间保持平衡。每个经理都认为自己的要求是最重要的；如果团队处理一个经理交代的任务，另一个就会感觉受了冷落。每个人压力都很大，因为他们面对的是一个两难的选择。

如果这个团队使用看板方法来把工作流程可视化，所有经理提出的功能请求都会变成看板第一栏中的工作项。如果请求的数量超出团队所能处理的上限，贴纸就会在那一栏中堆积，所有人都能看得到。现在团队知道该怎么做了：让所有经理就该栏的工作上限达成一个共识。

单就这个例子，我们不妨假设他们达成了共识并设置了一个工作上限。经理会像往常一样添加新的功能请求，在达到工作上限之前一切都照常不变。不过一旦某个经理达到了工作上限，有意思的事情就发生了：她不能再往看板上贴贴纸了。要想贴新的贴纸，必须得先从已经贴在看板上的帖纸中选一个揭下来。如果这个经理不想揭掉自己的贴纸，她就不得不跟其他经理商量，看是否有人愿意给她腾出一张贴纸的空间。

值得反复强调的是：当经理碰到工作上限时，她不会埋怨开发团队。她把这看作是系统的一个局限，并跟其他经理一起找出解决方案。这是系统思维很重要的一部分。当每个处于系统中的人都能够意识到系统的存在，他们就会在系统内寻求解决问题的方案。而且因为每个人都理解该系统，不均衡和超复杂就不再仅仅是开发团队的问题。它们是所有人的问题，包括经理在内。

新的行为方式就这样在团队的外部自然出现。现在经理不会把超负荷的情况怪到开发团队身上（这种情况不一定是团队的错），它们变得对看板第一栏中的贴纸更加关注了，因为这些贴纸就是团队将要处理的工作任务。他们可能会召开讨论优先级的会议，或者在他们自己中间进行配额交易，或者寻找其他方法来决定哪些任务可以进入队列。但是有一件事不再发生了：开发团队不再受到责难，因为它不再被要求承担它根本不可能承担的任务量。

换句话说，在使用看板方法之前，开发团队需要与有着神奇思维的经理打交道，后者以为可以把无限量的工作交给团队去完成。这些经理总是对结果失望，并感觉团队的承诺都没能兑现。有了工作上限和明确的规则，那种神奇思维就被消除了。经理改变了他们的行为，不是因为有人要求他们这么做，而是因为他们在一个系统中工作，而该系统鼓励他们以一种不同的行为方式来做事。结果是，团队有足够的缓冲空间来以放松的心态和正确的方式完成工作。原本用于跟个别经理讨价还价的全部精力现在都可以完全放在工作上了。这是一种更高效也更愉快的工作方式。

要点回顾

- 看板团队的一个目标是最大化**工作流量**，或者说最大化工作项移出系统的速度。
- 看板方法的**测量并管理工作流**量实践意味着对工作流量进行测量并对流程进行调整，以达到最大的工作流量。
- **累积流量图**就是一个工作进度面积图，但额外展示了每天新添加到工作流中的工作项的数量（即**到达速度**）、工作流中全部工作项的数量（即**工作存量**）以及每个工作项在系统中存在的平均时间（即**交付时间**）。
- 当到达速度和工作存量不随时间变化时，系统就是**稳定**的；看板团队通过设置进行中工作上限来让系统稳定。
- 如果系统是稳定的，**利特尔法则**就适用于该系统，也就是说平均交付时间总是等于长期到达速度乘以长期工作存量。
- 如果你的团队能够通过设置工作上限来让工作流程稳定下来，你可以让你的利益干系人通过同意不增加新工作项的方法来减少用户等待时间，原因是这种方法能够降低到达速度。
- 看板团队常常会把**流程规则明确化**，做法是给看板的每一栏底部增加完成定义或退出条件。
- 当看板团队通过增加工作上限、管理工作流量以及将流程规则明确化等方法逐渐改进整个系统时，公司的其他部分常常会自然地产生一些改进了的行为方式。

常见问题

当你们谈到进行中工作上限和明确的规则时，好像它们能够完全改变整个团队的作业方式。这听起来有点牵强。真的是这样吗？

答案也许让人吃惊，不过确实是这样的。要理解这背后的原因，可能回到精益和看板方法的源头会有所帮助。

工作上限是让流程变得顺畅的一个简单却十分有效的工具，它的这种效果早就已经为人们所熟知了。看板方法基于 20 世纪 50 年代源于丰田的一套系统（就像精益的很多思想和工具一样）。看板这个名称来自日本语中意为信号卡（就是签字板或者告示牌）的词。在生产车间里，装配线上的一个岗位会分配到一定数量的看板（即一张张写有零件编号或名称的实体卡片），该岗位需要用到的每一种零件都会分配对应的看板。当某一个零件发生短缺需要更多供给时，就把该零件对应的看板放到一个空的手推车中。负责配送零件的工人会向这个手推车中放入零件，每个看板对应一个零件，并且会把看板系在零件上。随着零件被使用，看板会再次回到手推车中。这就是装配线上的工人如何拉取他

们需要的零件，这种方法让整个公司可以减少零件的总库存。在这里限制每个零件的看板数量其实就是设置了工作上限。

对于开发人员来说，有时候不容易看出这种使用看板卡片的拉取式系统可以被应用到软件开发上。程序员不喜欢将自己与装配线上的工人相提并论，这也合理，因为开发工作更类似于汽车工程师和设计师的工作而不是装配工人的工作。但是看板方法并不是说要把人当作装配线上的工人，或者当作机器中的齿轮。看板方法与装配线的共同点在于它们使用看板（零件手推车中的卡片和白板上的贴纸）来表示有更多的工作准备进行了。

有时候使用一个跟装配线或者软件开发完全不相关的例子可以让这个类比更加容易理解。每个夏天，在圣保罗的一个大型集市上都会举行明尼苏达州博览会，参加该项活动的明尼苏达人超过一百万。集市上散布着很多出售食品的商家。最受欢迎的商家之一是一个大型的烤玉米摊位。有些其他受欢迎的商家排了很长的队，比如为了买一串炸橄榄而排在一个有二十多人的队伍，并不是什么稀奇事[4]。但是这个烤玉米摊位却能够服务大量的人流，通常只需等待很短的时间，而且基本不需要排队。这是如何做到的呢？

炸橄榄那家有两个完全相同的窗口，常常都排着很长的队，卖烤玉米这家店有两个不同的窗口。在第一个窗口，你交上钱，然后领一张票。接着，你拿着票到第二个窗口，把票交过去，然后拿一穗烤玉米走。

对于第一次逛这种集市的人来说，这种做法似乎有点不必要：我把钱给了一个人，拿了一张票，然后马上就把票交给另一个人，何必呢？但是现在你应该能够认出这个额外的步骤了：那张票就是一个看板，而烤玉米店就是用它设置了一个工作上限，从而建立起一个拉取式系统。

在第二个窗口烤玉米的人知道一次发出去了多少张票，而且他只会交给第一个窗口卖票的那个人适当数量的票，从而不至于让很多人拿着票在队伍中等玉米烤出来。这就使得他们能够根据情况来调整烤架上的玉米数量，在人少的时候，就不往烤架上放玉米，以免玉米烤糊，而高峰时段可以在烤架上放很多玉米，从而让队伍尽可能短。这还意味着他们可以雇用额外的人手专门负责烤，于是有更多的人手可以专注于耗时的“烤”这个任务，而不必停下来收钱。在不常见的供小于求的局面下，只有第一个窗口那里才形成等待的队伍。烤玉米的人则不需要面对很多人排着队抢着交钱的那种混乱局面，这使得他们可以专注于烤玉米，从而让队伍能够很快地缩短，然后消失。而且因为工作进行得非常顺畅，烤玉米的人能够进入一种工作节奏，使得他们感觉更加舒服，从而也做得更好。

调整工作上限帮助烤玉米店处理客流量的变化。在高峰时段（比如旁边有大型的表演开始了）增加发放出去的票的数量和烤架上玉米的数量，他们就能够顺畅并快速地提高产量。无论卖玉米的商家是否知道这一点，但他们已经在利用利特尔法则了：通过显示到达系统的人的数量，他们可以减少已经交了钱并正在等待玉米的顾客的等待时间。（你能想象出这在累积流量图上是什么样子吗？）

注 4：截止到 2014 年，卖炸橄榄的那家店使用的是典型推动式系统（排着长队的人们）。本书出版后那家店可能已经换用了看板系统。

这个例子可以帮助说明推动式系统和拉取式系统的区别。炸橄榄摊位使用的是推动式系统：很多顾客排成长队。顾客处于“推动”的状态，因为随着他们排起长队，他们就创造了需求。卖橄榄的商家对于这种需求完全无法掌控；他们只能把橄榄卖给队伍中的下一个人。而在卖玉米的店家这里，烤玉米的窗口那个人是在“拉取”，因为收钱的窗口的票是他们给过去的。他们通过这些票把人“拉”到烤玉米的窗口，而且他们可以通过限制发出去的票的数量来控制拉过来的人的数量。这就是他们设置工作上限的方式，这让他们能够将烤架上的玉米与发放出去的票数匹配起来。对这个工作上限的调整可以让他们随时应对客流的变化。

我不是装配汽车的，我也不是烤玉米的。我是做软件开发的，所以我一直都在解决不同的问题。你说的这些怎么应用到我的身上？

看板方法一样适用于你，因为它只是一个让你能够处理随机应对项目变化的方法。限制进行中的工作和控制队列的大小能够减少变化。将注意力集中在延期的源头和像瓶颈聚类分析这样的技巧（或者根本原因分析和那些设计用来减少瓶颈的可能性和影响的改进）可以减少变化。消除变化的根源并反过来影响经济后果以及系统的风险管理，这是看板方法的核心。[5]

这是开发团队需要做的事而且需要多做。事实上，你可以说软件团队其实需要处理比汽车制造商或烤玉米店家更多的变化。软件不同于世界上任何一种工程制品的一点在于它是可变的。因为它没有一个实体，软件工程师可以比其他工程师在更大范围上修改软件的形态，而且这种修改可以在项目更晚的阶段实施。但是每一处修改都是有风险的，如果你的项目在变化性的作用下改来改去，就会导致 Mura（不均衡），因为那些修改变得具有破坏性了。

那么你能够做什么来保证这种不均衡不会对项目造成破坏呢？

大部分瀑布式团队所使用的传统思路里，这个问题可以通过设置一个变更控制流程来解决。对软件的变更被控制着，被放慢了，而且如果可能的话，尽量地避免掉。对计划的修改需要首先被提议，然后由团队进行考虑，对它们对项目的影响进行分析，并且必须得到各方面的一致同意才能付诸实施。这是避免项目受变化性影响的一个有效方法，它也确实降低了项目的风险，但是它同时也保证了你的团队开发的软件是基于原本的计划，不管计划中的东西是否确实对用户有价值。传统项目管理的世界并不像刚才我们说的那么没有希望。不过我们在第 6 章和第 7 章中已经看到了，如果给团队一个放松的环境，在这个环境里能够保证适当的工作时间，并且有足够的精力去思考要解决的问题，团队是能够开发出容易修改的软件的。优秀的软件项目经理不是傻子，他们能够认识到这一点。这就是为什么他们会在日程中加入缓冲性的任务，就像极限编程团队所做的那样。但是变更控制流程意味着不能选择极限编程中的那种添加可以推到下一个迭代的低优先级任务的方法，因为那样的话就需要修改计划，而这个修改必须经过审核和批准。所以，他们会增加缓冲区，或者说增加没有规定具体工作内容的空任务，专门用来在日程中占位，他们常常使用数学公式来决定在项目的不同节点应该添加多少这种占位用的缓冲区。这样他们就可以按照原始的计划走，并

注 5：感谢 David Anderson 帮助我们解释看板方法如何处理项目中的变化。

给予团队足够的时间来完成它的工作，同时依然能够去除额外的变化，以便项目能够按时完成。

换一种说法，当使用传统项目管理方法的团队面临棘手的问题时，它的反应通常是给项目日程增加缓冲，不管问题的根源是什么。

看板团队并不需要往项目日程中添加缓冲，因为这么做会把变化性隐藏起来，尝试作出好的决定的人们无法看见。精益思维很重视要着眼全局，而任何对信息的隐藏都让我们无法看清全局。精益思维还告诉我们，当存在因不均衡而导致的浪费时，我们必须找出问题的根本原因并修复它。看板团队不会用缓冲的方法把不均衡隐藏起来，而会在将工作流程可视化的时候把它暴露出来，然后引入工作上限和其他的规则来解决它。它把变化性及其根本原因暴露到了太阳底下。

如果我没法说服我的上司同意限制进行中的工作呢？

那样的话你就没法实施看板方法了。当你看到看板方法是如何起作用的时候，貌似设置队列并限制流量能够帮助你给团队带来很可观的改进。不过，一切都始于设置工作上限，而这要求你获得你的经理（或者常常是好几个经理）的同意，在团队达到工作上限的时候停止增加新的工作。

这可能会成为一次改进努力的阿喀琉斯之踵，尤其是当你的上司有着神奇思维的时候。比方说你花了数周甚至数月的时间帮助团队理解看板方法，一起把工作流程可视化，创建看板，并且进行了仔细的量化测量并找出了合适的工作上限。现在你把你的上司和其他一些经理聚到一起，并自豪地向他们展示你的提案。你的上司把提案完整地看了一遍，并进行了仔细的考虑。最后他说："看板不错，这些量化指标也非常好，我完全同意这个方案。我只需要你做一个小小的调整，把看板栏目上方的那个数字去掉。"

对你的上司来说，似乎他作出了尽量小的妥协：只不过是去掉一个小小的数字。但是工作上限正是看板方法背后的秘密武器。没有它，团队就没法限制工作流量，而看板方法也就没法起到它的效果。

让某个阶段的工作慢下来却能够使得整个项目快起来，这一点似乎不是那么符合直觉。而且即使你已经把工作流程可视化了，进行了量化，而且也找出了由不均衡、超负荷以及不合理或不可能的工作而导致的浪费，这些对你的上司来讲可能也不见得有足够的说服力。如果确实说服力不足的话，你的上司就会注意到工作上限，并想要在这一点上进行折中。在他的感觉中，去掉这种限制会帮助团队优化它的工作，并防止人力资源被闲置。他很可能感觉设置工作上限会阻碍工作的完成，而他则是在好心地给团队指出这一点。他还可能看到了工作上限会给团队留出缓冲空间，并且害怕团队会滥用它，把工作慢下来，用省下来的时间去休息。或者（往好的方向想）他可能甚至不太清楚为什么他不喜欢工作上限这个东西，只是感觉这东西不好。或者他可能完全是出于理性而根本没有神奇思维：他可能意识到某种交付节奏会偶尔导致某一个特性从一个版本推迟到下一个版本，而当他的用户发现时，会受不了，并开始大炒鱿鱼。但是不管你的老板脑子里想了什么，把工作上限去掉相当于去掉了看板方法的支柱。

那么如果你无法在工作上限这一点上得到经理的同意，你该怎么办呢？

这种情况是考虑尝试 Scrum 的一个好机会，因为 Scrum 有可预见的时间约束和功能范围，这些都会提前规定好，并且由一个专职的产品所有者来管理。

Scrum 之所以在经理面临限制时能够有效，一个原因是积压工作表对于项目来讲是内部的。Scrum 团队依赖产品所有者来决定哪些工作项应该添加到积压工作表中，哪些应该包含在每次冲刺中，并且这个产品所有者是属于团队的一员。这样一来团队就可以控制开发的范围，而经理和用户可以与产品所有者一起来消化开发范围中的修改和变化。与传统的变更控制流程相比，这给了团队更大的灵活性，但是依然能够限制公司面临的变化。团队甚至也使用一个白板来把工作流程可视化，不过这个白板是一个任务板，而不是看板，因为它包含的是具体的任务而不是工作项，而且它上头没有工作上限和其他成文的规则。

另一方面，看板方法中的队列和缓冲区对项目来讲是外部的。一旦同意设置一个工作上限，哪些任务该进入队列就是由客户自己（包括经理、用户和产品所有者等）来决定了。他们必须要同意设置工作上限，而且不再与变化隔离开来，不过反过来他们也对应该开发哪些功能和以什么顺序进行开发有了更多的控制权。用制造业的术语说，这叫作“即时”交付，因为开发团队可以根据工作流程各阶段中等待处理的工作项（看板上各栏中的贴纸）的多少来决定接下来做哪一项工作，而这些决定可以在项目的后期才作出，甚至在团队开始做某个工作之前几分钟才作出。

如果说你的团队本来打算采用看板方法，可最后却变成了采用 Scrum，这也算是不错的成果。一旦你能让团队在一个稳定的系统下持续地工作，就可以帮助团队理解对流程进行改进到底意味着什么。如果你发现团队得到的是“聊胜于无”的效果，这也是在向着正确的方向前进，你可以继续沿着这个方向努力。而使用看板方法来改进团队的软件开发方式是很好的继续前进的方法。

看板方法声称自己不是一个管理项目的系统，但是当你把工作项在看板上挪来挪去的时候，它看起来确实很像一个项目管理系统啊。你确定看板方法不是一个项目管理系统？

是的，看板方法肯定不是一个项目管理系统，也不是一种软件开发方法。看板方法是一个过程改善方法。

在你改善一个过程之前，首先需要理解它。看板是帮助你理解你正在使用的软件开发系统的一个非常棒的工具，因为它是将工作流程可视化的非常有效的方法。

看板方法的这个特点可以有效地避开很多过程改善努力都常常陷入的一个致命陷阱：他们并不真正从全面理解他们团队当前的工作方法开始。典型的过程改进努力常常始于一群经理和项目负责人聚到一起，创建出当前软件过程的图示或者文字描述。他们通常知道过程改善需要一个起点，所以会发一些调查问卷，跟软件团队座谈，并用其他的办法来获取信息。然后他们会把所有信息整理成某种成文的过程，并用它作为改进的起点。

问题是像这样收集信息是非常困难的，而且很少能反映团队事实上是如何开发软件的。假如说你是一个团队成员，有一个由高级经理组成的委员会问你项目是如何运转的。你会去强调问题和失败吗？还是说你会试着粉饰太平？即使你尽力给出你能给出的最准确

答案，我们也还是倾向于更容易回想起成功的时刻，而对我们面临的挑战和失败不会记得那么深刻。

这就是看板方法更擅长的地方。在使用看板方法的一开始，团队就通过创建一块看板的方式把工作流程可视化了。但它并不是到这就完了，随着项目的推进，它会用工作项来更新看板。团队中的每个人都理解他们这么做并不是为了管理项目，而是因为他们在尝试精确地描绘出他们用以开发软件的那个系统。他们的描绘是非常精确的，因为他们会保持看板时刻反映最新的情况。同时，这种描绘也是非常有用的，当他们想要找到接下来要做的故事或特性时，可以清楚地看到哪些工作正在系统中流动。但是他们不会使用看板来决定他们要具体执行哪些任务。看板上的所有信息都是工作项一级的，而不是任务一级的。如果他们的系统中还用到了任务板、甘特图或者其他什么项目管理方法，他们就用那种方法来决定执行哪些具体任务。

正因为看板方法具备这种有效地将工作流程可视化（通过看板）和量化工作流量（通过CFD）的方法，所以团队可以找出最有效的改进方式。但这并不是使用看板方法的团队取得大量过程改善方面成功的唯一原因。看板方法能够取得最好效果的前提是团队具备精益思维，而正如我们在 Scrum 和极限编程中看到的，帮助你的团队进入精益思维的一个有效方法就是开始使用看板方法的实践。一旦他们开始可视化工作流程并量化工作流量，他们就将开始理解系统中浪费的所在，并使用选择思维来给自己创造更多选择，同时学着看清整个系统。

现在就可以做的事

下面是你现在就可以自己或与团队一起尝试做的事情。

- 如果你今天就要创建一个看板，看板上会有哪些栏目？如果你还没有创建过价值流示意图，现在是画一个价值流图的好时机，它可以告诉你看板上可能有哪些栏。
- 你刚创建的看板上有没有那一列比较适合设置一个工作上限？专门给这一列贴上一些工作项贴纸。这些工作项中，事实上正在执行中的有多少个？你该把工作上限定为多少？
- 搞清楚要设置一个工作上限你需要跟那些人商量。
- 创建一个简单的看板并用它跟踪你的项目一个星期。基于你的数据创建一个 CFD。你的工作存量稳定吗？到达速度呢？

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 关于看板方法：《看板方法 : 科技企业渐进变革成功之道》，David J. Anderson 著。
- 关于系统思维：《敏捷软件开发工具》，Mary Poppendieck 和 Tom Popendieck 著。

教练技巧

下面是帮助团队理解本章思想的敏捷教练技巧。

- 看板方法始于精益思维，所以帮助你的团队使用看板方法的第一步是帮助它掌握精益思维。第 8 章的教练技巧会有所帮助。
- 对于那些想要使用看板来改进其流程的团队，最大的一个障碍是理解看板方法不是一个项目管理系统。作为教练，你的工作是帮助团队认识到什么才能算是项目管理系统，以及看板方法如何帮助团队理解其系统，而不是管理它的项目。
- 帮助你的团队开始看到它什么时候过早地作出承诺,以及那样会如何导致项目中的浪费。帮助它辨别选项，并学习在最后责任时刻作出决定。人们（尤其是紧张的经理或者有着神奇思维的经理）常常会要求团队承诺一个具体的日期。你需要帮助团队学会只做那些必要的承诺。
- 看板方法也要求团队具备系统思维。这是精益很重要的一部分，而且它也是让看板方法能够有效的基本思想之一。为了让大家能够在认识中把机器的齿轮和让机器运转的人区分开，帮助团队用看板把工作流程可视化常常是很好的第一步。

第 10 章

敏捷教练

你已经学习了 Scrum、极限编程、精益和看板方法，知道了它们的共同之处，了解了它们要达到的目的。如果你与别人合作开发软件，那么至少看到了能够帮助你的团队的一些东西，包括实践、思想和态度变化等。

很好！现在，开始去做吧。让你的团队敏捷起来。马上！

这似乎还不是很现实，是吧？从书本上看到一些价值观、原则、思维方式和实践，与实际去改变一个团队的工作方式，这两者之间存在着很大的不同。

有些团队能够拿来一本 Scrum 或者极限编程的书，采用其中的实践，并马上看到非常好的效果。读过本书的前九章之后，你应该能够知道这是为什么：那些团队已经具备了与敏捷宣言及其理念相一致的思维方式。对于这样的团队，采用敏捷方法感觉很容易，因为团队中的个人并未改变他们对工作的看法。所以，如果你已经具备了与你所想要采用的敏捷方法相一致的思维方式，会更有可能取得成功。

可是如果你还没有具备一种能够与 Scrum、极限编程或其他敏捷方法相兼容的思维方式呢？如果你工作的环境中很难用敏捷的价值观取得成功呢？如果单打独斗在你的团队中比团队合作得到的回报多得多呢？如果错误会受到严厉的惩罚呢？如果你的工作环境扼杀创新，或者让你的团队无法接触到客户、用户或其他能够帮助你们理解要开发的软件的人呢？这些都会妨碍你采用敏捷方法。

这时就需要*敏捷教练*的介入。敏捷教练就是帮助一个团队采用敏捷方法的人。他将帮助团队中的每个人学会一种新的态度和思维方式，并克服精神上的、感情上的，还有技术上的那些阻碍团队采用敏捷方法的障碍。敏捷教练与团队中的每个人合作，不仅帮助后者理解“怎么样”去实施他们要采用的新实践，而且帮助后者理解“为什么”要这样做。任何人被要求在工作中尝试新东西的时候，面对变化都会产生自然的排斥，甚至畏惧情绪，教练

将帮助团队克服这些情绪。

我们已经在本书中看到了很多只得到“聊胜于无”效果的例子：一个团队采用了某种敏捷方法的实践，但是团队成员只得到了微小的改进，因为他们并未真正改变他们对工作的认识，或者并未真正改变他们对合作开发软件的态度。换句话说，要想用某个敏捷方法取得好的效果，你的团队需要具备敏捷的思维方式。敏捷宣言中的敏捷价值观和原则可以帮助团队进入正确的思维，出于同样的原因，每一种敏捷方法也都有着自己的价值观和原则。当每个人都能进入一种与敏捷的价值观和原则以及他们所采用的具体方法相适应的思维时，团队就能在采用敏捷方法上取得最佳的效果。

敏捷教练的目标是帮助团队获得更好更敏捷的思维。一个好的教练会帮助团队选择一套与它已有的思维相适应的方法，并以一种适合团队的方式来给它介绍相应的价值观、原则和实践。教练会帮助团队采用其实践，然后使用这些实践来帮助团队学习并内化价值观和原则，以此来慢慢改变团队成员的态度，同时使他们进入正确的思维方式，从而得到比“聊胜于无”更好的结果。

在本章中，你将学习如何进行敏捷指导：团队是如何学习的，敏捷教练如何帮助团队改变其思维从而让它能够更容易地采用一套敏捷方法，以及教练如何帮助你的团队变得更加敏捷。



故事：有一个正在开发一个手机相机应用的团队
（团队所在的公司刚刚被某大型互联网集团公司收购）

- Catherine——一位开发人员
- Timothy——另一位开发人员
- Dan——老板

10.1　第3幕：还有一件事（又来了?!）……

“嘿，Cathy！ 你有时间吗？我有个好消息。”

Catherine 在老板叫她去他的办公室时，暗暗叹了口气，暗想，我永远也没法习惯他的这句话。坐下来的时候，Catherine 强打着精神等待所谓的好消息。

“这个什么敏捷的玩意儿，你们干得很漂亮。真的，非常棒。”

Catherine 回想了一下她和 Timothy 使用看板方法来改进流程的这过去八个月。她找出了流程中的几处瓶颈，尤其是 Dan 和其他经理动不动就给他们分配新功能这个严重的瓶颈。她添加了一个队列之后，让人惊喜的事情发生了：随着经理开始互相争抢队列中的位置，他们慢慢不再要求 Catherine 和 Timothy 承担力所不能及的工作量了。

事实上，她想到这里，发现这是好几个月以来 Dan 第一次因为“好消息”把她叫过来，而之前所谓的“好消息”都意味着 Dan 有一个重要的最后时刻的修改，可能会完全扰乱她的项目。

“你能满意我很高兴，Dan。”Catherine 说。她其实是半带挖苦地说这句话的。

Dan 接着说：“而且我们母公司的人注意到了。他们了解了一下这个什么敏捷方法，既然我们已经取得了一些成绩，他们想让我们去教一教其他团队怎么做。”

Catherine 说：“等一下，你是在让我……啊？指导他们？”

“正是。”Dan 说。

“我不知道该怎么指导别人啊。”Catherine 回答说。

Dan 直勾勾地盯着她的眼睛，说：“这由不得你，Cathy。加油，鼓起劲儿来好好干。”

10.2 教练要理解人们为什么不想改变

> 你所在组织中的大多数人都在试图做好他们的工作。他们希望同事和上级看到他们擅长做那些分配给他们的任务。当一个人已经对他的工作适应并熟悉了的时候，他最不希望看到的就是有人过来让他采用一套全新的工作方式。
>
> ——Andrew Stellman，Jennifer Greene，《实用软件项目管理》

敏捷教练的大部分时间花在帮助团队中的人们改变他们的工作方式。这对教练和团队两方面都是一个挑战，因为只有教练能够看到全局。对于团队中的人们来说，他们被要求采用一套新的工作方法，但是他们不一定知道为什么要这么做。

很多情况下，一个带着好的意图想要采用某项实践的团队会对该实践进行修改，结果导致这个实践不再有效。比如，我们在第 5 章中看到，人们经常把每日站立会议变成了每天召开的进度报告会。每日站立会议的一个重要目标就是用自组织代替上令下行式的项目管理方式；每个人在会上问那三个问题的目的是让团队能够掌控自己的项目计划。但是很多尝试采用 Scrum 的团队最终不过是把它当作一个供团队成员每天汇报各自工作进度的会议，在这个会上 Scrum 主管实际上变成了给大家分配任务的项目经理。

类似地，有些团队在业务需求文档中加入了用户故事，但是却像大部分瀑布式团队那样把那些文档当作是规格文档。这些团队还是在做大需求先行式的开发，只不过加上了用户故事。还有，有些试图采用极限编程的团队并没有真正去做测试驱动开发，而是确保在代码写完后才编写的测试有很好的覆盖率，这说明测试对于设计没有任何影响，因为写测试的时候软件的设计早就定型了。

在所有这些情形中，团队中的人都是真诚地尝试去接受敏捷的。但是在每一个例子中，他们都没有真正理解这些实践只是一种理念在更大的生态系统中的一个组成部分。所以，他们没有尝试去改变他们的工作方式，而是专注于该项实践中他们感到熟悉的那部分。而我们怎么能期待他们不这么做呢？一个仅了解过上令下行式项目管理方法的团队，它完全没有自组织的经验，也没有相应的环境让它对每日站立会议的理解超出它的认知范围。

平心而论，大部分正在尝试接纳敏捷方法的团队已经在开发软件了，而且也取得了一些成功。（再说，完全机能失调的团队也很少会有一个允许它尝试敏捷的开明领导。）它们

只是自然而然地寻求做出小幅度的、增量式的改变，因为它们不想破坏已经在起作用的工作方法。

这就导致了在接纳敏捷的过程中最大的障碍之一，即团队成员会想："我也算见识过敏捷了，已经采用了那些我熟悉的实践，我的团队也比以前有了进步，这对我来说就足够了。"这就是我们所说的"聊胜于无"的情况（它也是导致很多人与敏捷失之交臂的原因），在这种情况下，敏捷被降格成了具体的、微小的改进，这与它原本应该带来的令人激动情景和当初的大肆宣传相去甚远。

为什么团队成员常常坚持只采用那些他们看起来熟悉的实践，而拒绝那些不能马上与他们现在的做法联系起来的其他实践呢？

因为每一个新实践都是一个变化，而任何变化都有失败的可能性。当这些变化导致工作上的失败时，人们是可能丢掉饭碗的。

每一个敏捷教练都应该时刻把这一点记在心头。教练工作是要帮助团队改变，而改变可能导致那些被要求改变的人们特别大的、却"完全理性的"情绪化反应。为什么呢？因为工作是人们赖以养家糊口的。

当我们在工作中被要求学习做新的事情时，我们不会感觉能够快速而轻易地掌握它们，这导致我们在情感上的严重不适应。我们脑子里的本能想法是："昨天我能顺利完成工作并养家糊口，但是今天我可不太确定了。"这就是被要求在工作中学习新东西可能导致焦虑和貌似不理性反应（在这种背景下，其实也算不上不理性）的一个重要原因。

人们抵触工作中的变化并不自觉地往他们已经熟悉的做法上靠的另外一个原因是，他们没有时间去仔细想清楚这背后的原因。比如，团队只在一个实验性项目上采用敏捷方法的情况十分常见。这常常始于一个人阅读了一本关于敏捷的书，并正在带领整个团队学习敏捷方法，但与此同时，他还是要面对各种截止时间、bug、冲突、需求变动以及一个典型项目中可能发生的所有其他事情。这可不是验证一种全新工作思路的理想场所。人们会尽可能地去做，但是如果有什么东西他们没有完全弄懂，他们会保留"敏捷"这个标签以及方法和实践的名称，但是他们的工作方式却几乎没有什么变化。旧的项目计划中的里程碑现在叫"冲刺"了，或者可能有人支起了一个任务板，却从未真正影响到进行中的工作。最后，团队依然是回到过去的老路上，因为它知道那种做法过去是有效的，而且还有截止时间在前面等着呢。

当一个团队使用敏捷实践的名称却没有改变它的工作方式时，不难看到为什么团队成员很快就对敏捷背后的思想大失所望了。对他们来说，认为敏捷不过是他们目前正在做的事情的另外一种叫法也是有其道理的。他们以为自己已经采用了敏捷方法，但是他们并未改变他们的工作方式，所以得到的结果跟以前没什么不同。

团队中的人们会认为敏捷方法根本没用，却没有意识到他们连敏捷方法的影子都没见过。他们有这种反应是因为他们被要求改变工作方式，却不理解为什么，而且没有人帮助他们在做出改变的同时保证新的实践和思想不至于变味。

这就是需要敏捷教练的原因。敏捷教练的一个重要工作就是在人们被要求做一件新的事情时，帮助他们去适应这种变化。教练需要给受训的人提供有关该变化的背景知识，不仅要

让他们了解新的变化是什么，还要让他们理解为什么。这会帮助团队切实地改变他们的工作方式，而不是简单地把敏捷方法中的名称照搬到他们已经存在的做法上。

比如，一个好教练会用每个人都能理解的语言解释每日站立会议如何帮助团队实现自组织，测试驱动开发如何帮助团队从功能角度思考并向实现增量式设计靠拢，或者用户故事如何帮助每个人理解软件用户的视角。教练帮助大家不仅仅形式上采用新的规则，而且还让大家开始看到他们前进的方向以及这些新做法最终能够给他们带来什么。

教练会留意团队抗拒变化的征兆

如果你曾经指导过很多团队，你将听到很多不同的人都重复一些相同的论调。下面列出了可能表明人们对变化感到不适应的一些说法，以及你能做些什么来应对，即使你不是一个敏捷教练，从教练的视角看这些问题也是有益处的。

- “我们现在的软件开发方法挺好的。你为什么要我使用一种不同的方法？”
  你很难与成功争论。如果你跟一个有着成功交付软件历史的团队一起工作，当然有权知道为什么需要改变，仅仅告知这是老板的要求是不够的，因为那将打击团队的士气。作为教练，你需要对团队过去的成绩保持正面的态度。但是，不要不好意思指出团队遇到的问题。每一种敏捷方法中的实践，其目的都是克服团队的问题；如果你能帮助团队理解为什么那些问题会发生，并给他们提供解决方案，他们会更有可能接受变化。

- “这么做风险太大了。”
  这是对敏捷方法的一个很常见的反应，尤其是那些习惯了上令下行式项目管理方法的人。怎么没有缓冲区呢？风险控制机制哪去了？还有那些让项目慢下来且能够保护我不担责任的额外官僚流程都哪去了？项目计划可以是不透明的，这能让团队感觉良好：除了模糊的里程碑，它不需要跟外部共享更多的细节，而且它可以创建一些缓冲来减少甚至消除变化性。当你使用进度报告作为软件进度的主要衡量标准时，你可以控制将哪些信息给到用户和客户。对于习惯了这些的团队，敏捷方法确实可能感觉很有风险。在第 3 章中，我们学习过“可工作的软件是衡量进度的首要标准”这一原则。如果一个敏捷团队中出了什么问题，每个人都会知道。作为教练，你的工作是帮助经理、用户和客户适应这种不掺假的进度指标，并给团队一个安全的环境，允许团队成员犯错，只要他们未来能够吸取经验教训即可。如果这在当下不现实，那么你的工作就是帮助所有人，尤其是老板，设定一个目标在未来改变团队的气氛和态度。

- “结对编程（或测试驱动开发，或其他某项实践）就是对我不起作用。”
  一个习惯了独立工作的开发人员心底里可能感觉结对编程会降低他的效率。这也许有他的道理，如果在他的团队（或者老板）的思维里，错误就是不能接受的，那么他对于旁边有人看着他编码而感觉到不舒服就是很正常的。任何曾经教过孩子开车的人都能理解那种一个资深开发人员让一个新手控制键盘而自己在一边看着的感觉。人们对这些实践感觉到不适的原因有很多，尤其是当团队的思维方式与该项实践不太匹配的时候更是这样。好的敏捷教练会帮助团队首先选择那些适合于其思维方式的实践。随着团队成员开始使用这些实践，他们会看到这些实践的好处，进而理解这些实践是如何工作，又是为什么能够工作的。有了优秀教练的指引，大家会自然地开始转向一种更敏捷的思维方式。

- “敏捷方法不适用于我们这种业务。”
  人们常常会这么说，因为他们习惯了在工作开始前先准备庞大而详细的计划和设计，而且他们无法想象以其他的方式工作。上令下行式的项目经理和老板更喜欢在工作开始前把功能范围、需求以及项目计划落到纸面上。同样，在第一行代码写下之前整个系统设计已经落实到纸面上，这种做法能让架构师和开发负责人感到安心。现在，他们被要求信任团队，并让后者在最后责任时刻才作出决定，而这意味着让他们放弃控制权。所以，他们会说类似这样的话：“我们的业务非常复杂。其他公司的人也许可以直接进入开发阶段，但是因为我们业务的要求，我们需要在一开始就做好计划和设计。”事实是，每一种业务都很复杂，而且每个项目都需要分析和计划。一个好的敏捷教练会帮助经理们、业务人员以及开发负责人看到：当团队被允许将项目分解成小块，并拥有在最后责任时刻作出决定的自由时，它能够更好地应对业务的复杂性。

- “这跟我们现在的做法没什么两样，不过是换了名称罢了。”
  这是人们拒绝使用敏捷方法的最常见原因。他们会想办法按照他们现有的方法做事，然后给它们配上他们读到过或听说过的敏捷实践的名称。这将使得敏捷方法基本失去作用，甚至变成错误的了，比如他们给某个让他们失败过的实践起了一个敏捷的名称。事实上，如果你在网上搜索一下“敏捷糟透了”，你会发现人们确确实实是这么干的：谴责敏捷方法不过是华而不实的炒作，因为它不过是老套的瀑布实践的又一个名称罢了。他们甚至会说敏捷是故意把简单的、常识性的软件开发复杂化了。作为教练，你需要认识到他们这么做并不是出于恶意；这只是某些从未亲眼见过（却自以为见过）敏捷方法的人们的一种自然反应。你作为教练的工作是帮助你的团队理解，敏捷确实与他们目前的做法不同，而且它也根本不糟糕。

10.3 教练要理解人们如何学习

> 掌握一样东西（如果有可能的话）的一个很好的模式来自武术。一个学武术的人要经过三个水平阶段，分别叫作守，破，离。守：按套路来；破：打破套路；离：自成一派。
>
> ——Lyssa Adkins，《如何构建敏捷项目管理团队》

> 《解析极限编程：拥抱变化》一书的作者 Kent Beck 也曾用类似的几个阶段来描述对极限编程的使用。当被问到极限编程与软件工程学院的“能力成熟模型”时，他提出了极限编程的三个成熟等级。
>
> (1) 按照既有的规范做事。
> (2) 做到第一点之后，尝试对规则做些变动。
> (3) 最后，完全不在乎你是不是在使用极限编程。
>
> ——Alistair Cockburn，《敏捷软件开发（原书第 2 版）》

在第 2 章中，你了解了人们对敏捷方法的不同看法。正如盲人摸象一样，不同的人最初都对“什么是敏捷方法？”这个问题给出了各自的答案。

程序员看到了极限编程中像测试驱动开发和增量式设计这类实践，于是认为敏捷方法是关于编程、设计和架构的。另一方面，项目经理在读到了 Scrum 的冲刺、计划、积压工作表

以及回顾会议之后，可能会认为敏捷方法是要改进项目管理方面的实践。

本书用了几百页的篇幅探讨了 Scrum、极限编程、精益和看板方法。我们也谈到了思维方式、价值观、原则和实践。我们还给你展示了如何把这些都结合起来，帮助你的团队为用户和客户创造更多的价值。我们帮助你找出了那些你今天就可以开始做的有助于你的团队的事情，并给你了一些工具（比如那个"你是否能接受……？"的小测验）来帮助你的团队改变思维方式。我们给了你很多学习敏捷方法和理解 Scrum、极限编程、精益和看板方法的工具。

可是，这并不是大部分人接触敏捷方法的方式。

如果全世界的每个开发人员都购买并阅读我们的书，我们当然求之不得，可事实是大部分团队中的大部分人不是通过阅读来学习敏捷的。他们是在实践中学习。而且把一个东西分成较小的、简单的小块做起来更容易。

换句话说，当大部分人第一次学习敏捷方法的时候，他们真正想要的是一个简单的规则列表，只要他们遵循这些规则，就能够让他们快速地开发出更好的软件。

如果你是初次接触敏捷教练这个行当，这可能会让你感受到挫败感。想象一下你正在尝试教一个团队如何有效地采用 Scrum。根据第 5 章的解读，不理解自组织和集体承诺的人就不算真正理解 Scrum。于是你花大把的时间给他们讲解 Scrum 的开放、勇气、专注、承诺和尊重的价值观，以及自组织是如何起作用的。但是团队却很失望，这些对他们来说都是抽象的概念。他们只想知道如何使用任务板和用户故事。你希望他们能"理解"Scrum；他们则铁了心只要"聊胜于无"的效果，而你对此却无计可施。这到底是怎么了？

关于如何教学，古人提倡因材施教。好的敏捷教练清楚人们是如何学习的，并给予适合他们的信息、指导和例子，从而帮助他们进入学习的下一个阶段。教练应该明白，人的改变不是一朝一夕的。

在《敏捷软件开发（原书第 2 版）》一书中，Alistair Cockburn 提到了一个关于学习的基本思想，叫作*守破离*（shuhari）。它对想要弄清楚团队所处阶段的敏捷教练来说是一个非常有价值的工具。守破离是从武术中借鉴过来的概念，不过它也是理解人是如何学习任何其他东西的一个好方法。

所谓守破离，是指学习分为三个阶段。第一阶段是所谓的守（shu，"遵守"或"观察"），说的是初次接触一个新思想或新方法的人的思维。他想要的是他可以遵守的简单规则。这是人们老是揪着实践不放的一个原因：团队可以很容易地就采用冲刺、结对编程、使用任务板等制定出规则来，而且每个人都能清楚地看到规则是否被遵守了。

规则对于在工作中学习新东西的成年人来讲尤为重要，因为跟学校里的孩子不同，成年人并不总是具有学习新东西的习惯。当你为正在学习一个新系统（比如某个敏捷方法）的人提出一些简单的规则让她去遵守时，相当于给了她一个起点。像 Scrum、极限编程，还有看板方法这些敏捷方法中的那些简单直接的实践，对于那些尚处在"守"这个阶段的人来讲很合适，因为这些实践足够简单，人们能够专心地把它们做好。通过给团队设置一个规则（如"我们每天早上 10:30 开每日站立会议，会上每个人要回答三个问题"），你可以帮助团队成员走上学习某项原则（"我们通过自组织不断反省并调整我们的计划"）的正轨。

但是，你作为教练的目标不是要将敏捷捧上天。敏捷狂热者指的是这样一种人，他们认定有唯一一种敏捷方法对所有团队都有效，并且高声地、强制性地要求别人使用这种方法。敏捷狂热者倾向于只专注于他们学过的那些规则，而停留在“守”的层次上。他们相信他们遇到的任何问题都可以通过他们所知道且宣扬的一条规则来解决。敏捷狂热者不会成为好教练，因为他们不去花时间弄清他们指导的那些人处在学习的哪一个阶段。

一个敏捷教练需要理解的绝不仅仅是简单的规则，这就要说到守破离的下一个阶段，*破*（ha，“分离”或“打破”）。在这个阶段里，人们已经理解了实践中的规则，并且能够开始理解并内化驱动它们的那些原则。贯穿本书，你已经学过了，让你的团队先采用某种方法的实践，并通过这些实践（以及大量的团队讨论）逐步理解其价值观和原则，这是一种有效地改变团队思维方式的方法。当团队中的人们一起达到了“破”的阶段，他们就开始从“聊胜于无”向前迈进，并获取到真正接受了敏捷思维的团队所能够获取到的真正的生产力提升。

我们没怎么讨论过最后一个阶段，*离*（ri，“独立”）。当你到了学习“离”这个阶段，你就已经对那些思想、价值观以及原则十分熟稔了。这个时候你对方法什么的已经不是那么在乎了，因为你已经超越它们了。当你面临的某个问题可以由某一项具体实践解决的时候，你和你的团队就会去采用那项实践。你并不会去在乎说它到底是 Scrum、极限编程、看板方法还是瀑布方法，这不重要，而且你的流程也不需要有一个专门的名称。不过，这跟没人尝试过敏捷方法的团队中的乱作一团可不是一回事。一个全员达到“离”这个层次的团队将是一个顺畅的、运行良好的团队，每个人都在按正确的方法做事。

Cockburn 指出，这种“无招胜有招”的说法在外行听来让人苦恼。在《敏捷软件开发（原书第 2 版）》一书中，他谈到了处在“离”阶段的人们说的一些话对仍处在“守”阶段的人来讲很难理解。

> *“怎么好使怎么做。”*
>
> *“如果你真的在那么做，就不会意识到自己在那么做了。”*
>
> *“只要一个方法有益处，那就使用它。”*
>
> 对于那些对敏捷已烂熟于胸的人，这些都正确无比。但对于正处于“破”阶段的人来说，就很让人困惑了。而对于那些寻找可以遵循的流程的人来说，则是毫无用处的。

一个好的敏捷教练的第一项工作就是跟团队的每个成员交谈，弄清每个人当前的学习阶段。对教练来说，与处在“离”阶段的团队合作是很容易的，因为每个团队成员都知道如何学习，而且也愿意去适应新的实践。

另一方面，处在“守”阶段的人想要的是清楚的规则。比如，在任务板上是使用贴纸还是索引卡？作为教练你当然知道这不重要。但是从没用过任务板的人并不知道这个东西是无足轻重的还是至关重要的，所以，你的工作是给他设定一个规则。后面你可以给他解释你为什么选择了贴纸，而索引卡片可以达到一样的效果。这二者是有点儿不同，比如，索引卡片两面都可以写字，但贴纸只有一面可以，而你可以通过二者的相似之处和不同之处来帮助团队成员学习它们是如何都能够满足相同的原则的。

使用“守破离”帮助团队学习某种方法的价值观

现在假设你还是那个为了让团队理解 Scrum 而充满挫败感的教练。“守破离”可以帮助你理解为什么人们在学习价值观的时候会感到失望，以及为什么他们想要了解最看得见摸得着的实践，比如任务板和用户故事。一个任务板或用户故事作为“守”阶段的概念，是不难讲清楚的：它就是一条简单直接的规则，你可以马上用到项目中。Scrum 中到处都是“守”一级的规则（“每天开会并回答三个问题”）；这就是为什么人们通过采用它的实践取得了很多成功的原因之一。

另一方面，像开放性和承诺这种价值观则属于“破”这个阶段：它们是决定你如何在一个系统中使用规则的抽象概念。自组织和集体承诺只有在每个人都真正理解并内化了这些价值观的时候才会发生。“守破离”帮助我们理解为什么团队需要首先使用那些表面上的实践，然后才能“剥离”具体的实践并开始理解开放性和承诺的真正含义。

这就是为什么采用 Scrum 的实践是帮助团队学习其价值的非常有效的第一步。一个好教练应该是耐心的，在每一次冲刺中等待时机来给团队讲授价值观。这类机会常常出现在两个不同视角的人看到 Scrum 这只大象的不同角度的时候。

比如，假设一个开发人员在冲刺中途发现一个特性无法完成了，接下来的每日站立会议上他让产品所有者把该特性推迟到下一次冲刺中。产品所有者生气了。她怕客户或经理会对此不满，所以要求开发人员“不惜代价”完成它，即便这意味着必须抄近路且无法完成所有的代码。这个问题存在一个“守”一级的答案：Scrum 的规则决定了团队在冲刺评审会上只能演示那些真正完成了的特性，不完整的工作必须推迟到下一次冲刺。敏捷教练清楚 Scrum 的规则，而且可以使用它们来解决这种纠纷。

不过一个优秀的敏捷教练还会看到这里有“破”这个层次的机会。通过识别割裂的视角（如我们在第 2 章中所学的），教练可以尝试使之变得完整起来。这时教练可以帮助产品所有者从开发人员的角度看问题，草草交差会欠下技术债务，这将让整个代码库更加难以维护。这些技术债务反过来会导致未来的冲刺花费更长的时间。因此，就算产品所有者忽视 Scrum 的规则而把那个特性交付给客户了，最后她得到的将是更糟糕的结果，因为将来的开发会进行得更慢。这是很重要的一课，它将帮助产品所有者更进一步地理解 Scrum 的“专注”价值观，以及一个专注于完成工作的团队如何以更快的速度交付质量更好的产品。

教练还可以帮助开发人员理解那个特性有多么重要，理解它如何为客户创造价值，以及理解对该价值观的更深入的理解如何在将来帮助开发人员与产品所有者更紧密地合作来找出进一步细分特性的方法，并以更快的速度为客户创造价值。这是重要的一课，它将帮助开发人员进一步理解 Scrum 的“承诺”价值观，还有当团队真的理解了什么东西对客户和用户有价值时，它能够更加高效。

上面的这个例子说明的是一个对敏捷方法有着深刻理解的教练怎样在团队中扮演一个有价值的角色。教练帮助团队理解并接受敏捷方法的实践和规则。而通过这些规则，教练可以帮助团队中的每个人理解敏捷的价值观，并使用这些价值观来帮助每个人进入一个更“敏捷”的思维。

10.4 教练清楚如何让一套方法起作用

在项目中，冲突和问题每天都在发生。比如，一个好的 Scrum 主管会把他的大部分时间都用来解决问题。大部分此类问题不一定是给团队讲授 Scrum 的好机会。

那么，在我们刚才举的那个例子中，那个教练是怎么知道在某个特定的冲突中蕴含着给团队讲授 Scrum 的“专注”和“承诺”这两个价值观的机会的呢？这个具体问题的哪一点让那个教练识别出了这种机会呢？

那个教练之所以意识到了机会的存在，是因为他不仅清楚 Scrum 团队怎么样使用迭代，而且还很清楚为什么使用迭代。他知道如果团队容忍那些并未真正完成的工作，那么一个有时限的迭代可能变得完全失去效率。没有那条规则，迭代就变成了简单的项目里程碑：每次迭代的范围变得越来越易变，而团队也不再必须在每次迭代结束时交付可工作的软件。

这打破了几项使敏捷方法得以起作用的重要价值观和原则。比如，教练清楚可用的软件是进度的首要指标，而在冲刺结束时交付不完整的软件会给客户和用户一个错误的进度值。教练还清楚，敏捷团队重视客户协作，但是他同时也清楚客户协作并不意味着客户永远是对的。真正的含义是，如果团队发现计划中确实存在问题，可以与客户协作来找到一个解决方法，创造最大的价值。他会意识到如果客户和产品所有者能够给开发人员施压，让后者把未完成的软件提交到冲刺审查会议上，团队就会变得对真正的工作进度有所保留，而最后客户协作会让位于团队与客户间对合同的讨价还价。

那个教练知道所有这些，因为他理解敏捷方法和 Scrum 的价值观和原则，而且知道这些价值观和原则如何驱动具体的实践。他清楚是什么让敏捷方法能够运转。一个 Scrum 教练理解集体承诺和自组织。一个极限编程教练理解拥抱变化和增量式设计。一个看板教练理解对团队流程的改进需要设置工作上限并用它们来控制工作流量，而当有人妨碍设置这些工作上限时，整个看板方法就会失效。

在《如何构建敏捷项目管理团队》一书中，Lyssa Adkins 就教练如何与团队一起解决问题提出了一个建议，我们在本书前面的部分提到过，但这里有必要再提一次。

> 要允许失败：当然，这并不是说在团队即将冲下悬崖的时候你应该袖手旁观。不过，要利用每次冲刺中的各种机会让它适当地失败。与把它保护起来相比，让团队成员共同经历失败并从失败中找到出路可以更好地锻炼他们。你以为会给他们带来不利影响的那些东西，也许恰恰对他们最有帮助。不信走着瞧。

这对每一个敏捷教练都是非常好的建议，尤其是那些有很强的“控制欲”（即控制团队学习和进度的各个方面的欲望）的人。一个好的教练应该知道有时候团队可以也应该失败，因为这是通往成功项目和团队最有效也最高效的道路。

这就是定期回顾非常有价值的地方。我们学习过 Scrum 和极限编程团队如何利用回顾来回头审视一个项目并找出改进的方法。对一个敏捷教练来说，这是很好的帮助团队从失败中学习的机会，让团队失败目的就是让团队成员从中学习。如果团队是因为没能足够好地执行某项实践而失败，教练可以帮助团队学习那些要想做得更好所需的技能。但是有时候失败是因为团队需要改变其思维方式，因为敏捷教练知道为什么敏捷方法和实践能够起作用，他 / 她就能利用团队的失败来帮助团队了解更多具体的价值观或原则，这些价值观和

原则能够帮助团队作出更好的决定并避免失败。团队就这样成长起来。

但是一个好教练也清楚什么时候不能允许一个团队失败。正如一座房子，其承重墙在装修的时候是不能破坏的，每一种方法也有它的“承重墙”，一旦改变就会让项目变得不健全。一个教练需要对敏捷方法达到“破”一级的理解：他对其作为一个系统有着深入的了解，而且知道它为什么能起作用。在第 8 章中，我们提到过一个上司，他要求团队去掉工作上限，这在无形之间就对整个使用看板方法的努力釜底抽薪了。尚处在“守”阶段的人也许就同意这种改动了，也许觉得至少还有一部分被采纳了，总比什么都没有强。而一个好的看板教练清楚，虽然有些东西（比如看板上具体的栏目）可以做调整，但工作上限却不能动，因为它是整个看板方法的关键。

敏捷教练工作中一个更困难的部分是，弄清楚应该给团队、老板以及客户做多少讲解，讲到什么程度。简单地对某个方法的规则的“守”一级的讲解足以让团队先把该方法用起来，但是并不足以帮助团队进入正确的思维方式并得到“聊胜于无”的效果。有时候给团队一组规则，然后团队照着做就可以了；另外一些时候，团队成员可能会感觉他们不过是用一堆强加给他们的敏捷规则替换了同样强加给他们的瀑布式规则。“破”一级的理解能够帮助团队看到为什么这些规则能够产出更好的软件；不过这也可能让团队成员感觉太玄妙，摸不着头脑。敏捷教练就是要帮助团队以它自己的节奏向着“破”的层次前进，并避免那些对团队没有帮助的抽象讲解。

10.5 进行敏捷指导时的原则

如果说有一件我们贯穿本书一直在强调的事情的话，那就是敏捷方法要求人们具备正确的思维方式，并且团队是通过对价值观和原则的学习来进入这种思维方式的。这就是为什么每一种敏捷方法都有自己的一套价值观，以及为什么团队成员只有在理解并内化了这些价值观的情况下才能挖掘出敏捷方法的最大潜力。

那么敏捷指导本身也有一系列的价值观也就不意外了。John Wooden 是 20 世纪 60 年代、70 年代 UCLA 男子篮球队的教练，他被普遍认为是体育史上最伟大的教练之一。在他的 *Practical Modern Basketball* 一书中，他提出了做教练的五个基本原则：勤奋（industriousness）、热情（enthusiasm）、状态（condition）、基础（fundamentals）和培养团队精神（development of team spirit）。这五项原则对敏捷教练同样有意义。

- 勤奋
  改变一个团队开发软件的方式意味着要在它从未尝试过的事情上努力工作。开发人员必须要考虑计划和测试，而不仅仅是编码。产品所有者需要清楚团队在做什么，而不仅仅是把功能要求丢给团队。Scrum 主管需要学习如何把控制权交给团队，同时依然能够参与到项目中去。这些都是新的技能，而学习任何新技能都需要勤奋的工作。

- 热情
  当你真心投入到工作中时，你会对新的做事方法感到兴奋，并进入忘我状态。也确实很值得激动，因为你正在解决那些过去让你头疼的问题。当每个人都带着热情进入敏捷方法的学习时，你得到的就不仅仅是更好的软件；你还得到了一个所有人都更有创新意识、更开心而且更兴奋于每天一起工作的团队。

- 状态

  当每个团队成员都擅长他 / 她所做的工作时，敏捷方法才会起作用。敏捷方法中有一个确实存在但又没有明言的假设，那就是每个人都有一种对技艺的自豪感：他们尝试编写他们能够编写出来的最好软件，并努力工作去做得更好，而且他们是由衷地由他们工作中的自豪感推动的。这就是敏捷团队的每一位成员不断努力工作提高技能，以便他能够带着最好的状态进入到项目开发中的原因。

- 基础

  Wooden 在他的书中写到，“再好的系统也无法克服对基础的糟糕执行。教练必须确定他绝不允许自己被一个复杂的系统弄得分了心”，对敏捷教练尤其如此。敏捷方法之所以能够起作用，是因为他的价值观简单直接，它的方法包含的实践都不复杂。这些是敏捷方法的基础，而一个好的教练会努力让团队专注于自身，并帮助团队保持形式方式简单。

- 培养团队精神

  我们探讨过自组织、整体团队、充满精力的工作以及给团队助力：这些都是敏捷团队互相帮助以及创建协作的、创新的工作环境的方法。反过来，一个敏捷教练需要留意那些主要专注于自己个人的表现、职业目标、履历或晋升机会的团队成员，并帮助他们改变态度。Wooden 清楚地说明了这种人对团队精神的影响，以及应该怎么办：“教练必须利用他所掌握的每一点心理学，并使用所有可能的方法来在他的队伍中培养一种团队精神。不要放过任何鼓励团队合作和无私奉献的机会，每一位团队成员必须要渴望，而不仅仅是愿意，为了团队的利益牺牲个人的荣誉。自私、嫉妒、自我中心以及互相指责将破坏掉团队精神并毁掉一个团队的前途。教练必须清楚这一点并时刻保持警觉，要在问题还在萌芽状态的时候就消灭掉，以便杜绝此类情况。”

这五项原则是敏捷教练的思维的一个坚实基础。成为优秀的敏捷教练的重要一步就是理解、内化并使用这些原则，就像你使用敏捷宣言和任何一种敏捷方法中的价值观一样。

要点回顾

- **敏捷教练**是帮助团队采用敏捷方法并帮助团队中的每个人学习新思维方式的人。
- 高效率的敏捷教练通过专注于实践中团队熟悉的部分来帮助团队成员克服心理上的不适应。
- 教练需要注意到团队对变化感到不适的那些征兆。
- 人们学习新系统或新思维方式有三个阶段：**守破离**。
- 初次接触某个思想的人处于第一阶段，即**守**，这时候她还在学习具体的规则，她常常对具体的指示更感兴趣。
- **破**这个阶段是指某个人开始认识到了更大的那个系统，并开始让自己的思维去适应这个系统。
- 进入**离**阶段，人们已经对系统的思想十分熟稔，比如，他们知道什么时候不需要遵守某些规则。
- 一个高效率的敏捷教练知道为什么一个系统能工作，并且理解规则存在的原因。

更多学习资源

下面是与本章讨论的思想相关的深入学习资源。

- 关于如何进行敏捷指导：《如何构建敏捷项目管理团队》，Lyssa Adkins 著。
- 关于守破离以及人们如何学习和适应新系统：《敏捷软件开发（原书第 2 版）》，Alistair Cockburn 著。
- 关于做教练的基础知识：*Practical Modern Basketball*，*3rd Edition* ，John Wooden 著。
- 关于为什么团队会抗拒变化以及如何帮助团队克服这种阻力：《实用软件项目管理》，Andrew Stellman 和 Jennifer Greene 著。

关于作者

Andrew Stellman 是一名开发人员、架构师、演讲家、敏捷教练和项目经理，同时也是开发更好的软件方面的专家。他专业从事软件开发二十余年，曾经参与设计大型实时后端系统的架构，管理过大型的跨国软件团队，在某知名投资银行担任过副总裁，并为包括微软、美国国家经济研究局、美国银行、Notre Dame 及 MIT 在内的公司、学校和企业集团提供过咨询服务。在这期间，他曾有幸与一些非常优秀的程序员共事，并认为他从这些优秀程序员身上学到了很多东西。

Jennifer Greene 是一名敏捷教练、开发经理、业务分析员、项目经理、测试工程师、演讲家，同时也是软件工程实践和原则方面的权威。她在包括媒体、金融和 IT 咨询领域有超过二十年的软件开发经验。她曾与有着杰出开发人员和测试工程师的团队共事，解决棘手的技术问题，并在这个过程中将她的职业生涯专注于发现并解决那些工作流程中惯常出现的问题。

关于封面

本书封面上的动物是一只节尾狨猴，也叫节尾猴。它是 Callimico 属的唯一成员，所以有时也称为 callimico。因为它独有的某些特点，有时候在进行归类时会将它与狨猴分入不同类别，比如它有三对臼齿，一次只生育一只幼崽，每个脚趾上都有爪子。瑞士的自然学家 Emil August Goeldi 在 20 世纪早期发现了这一物种，所以这个物种以他的名字命名。

这种猴子栖息在亚马逊河流域的上游，在巴西、哥伦比亚、贝鲁、玻利维亚和厄瓜多尔境内。它们一般生活在森林的“底层”，常见于森林、河流及竹子栖息地的低矮灌木中。它们最喜欢的食物有昆虫、蘑菇（在旱季的时候）和水果（在雨季的时候）。它们以大约六只一群的规模群居，而且不喜欢分开，所以它们通过一种高亢的叫声保持联络。它们通过垂直的跳跃在树木间游走，不过他们白天的大部分时间都躺在一团树叶里睡觉。

它们的体长大约 20~23 厘米，比松鼠大一点，但是他们的尾巴可能长达 30 厘米。它们脸上的皮肤发黑，毛则是黑色或者深棕色，常伴有稍浅的高亮。雌性在 8.5 个月时达到性成熟；雄性则需 16.5 个月。雌性的数量约为雄性的两倍，且每年能够生育两次。幼崽出生大约 3 周之后，就主要由父亲照顾了。

节尾猴的寿命可长达 20 年，至少圈养情况下是这样。但它们是濒危物种，而且它们比普通狨猴要稀少得多。计划中的建设以及其他栖息地变化在威胁着它们的种群和生存。目前已经很少能在野外看到它们。

O’Reilly 很多书籍封面上的动物都处于濒危状态；这些动物对我们的世界来说都很重要。想知道你能够做什么来帮助它们，请访问 animals.oreilly.com（http://animals.oreilly.com/）。

封面图片出自 Lydekker 的 *Royal Natural History*。

O'REILLY®

学习敏捷：构建高效团队

敏捷方法革新了软件团队的开发方式。然而，目前的敏捷实践门类众多，很多团队不知如何选择。本书立足实际情况，帮助准备接受敏捷的团队理清头绪。作者首先介绍敏捷方法背后的原则，进而详细讲解四种常用的敏捷方法：Scrum、极限编程、精益和看板方法。

上述每一种方法都侧重于开发过程中的一个不同方面，但目标都是改变团队的思维方式，将服从计划的独立个体凝聚成共同决策的团队。无论是初次接触还是重新尝试，你和你的团队都可以借助本书学习如何选择最合适的敏捷方法。

- 理解敏捷价值观和原则的初衷
- 理解Scrum为何强调项目管理、自组织和集体承诺
- 通过测试先行、结对编程等极限编程实践聚焦软件设计和架构
- 使用精益思维给团队增添力量，避免浪费并快速交付软件
- 学习看板方法如何在实践中管理流程，协助交付优秀的软件
- 在教练指导下采用敏捷的实践和原则

Andrew Stellman和**Jennifer Greene**，1998年进入软件开发行业，并一直撰写软件工程相关文章。他们在2003年共同创立了Stellman & Greene咨询公司，同时仍每天与软件团队并肩战斗，为用户开发和交付软件。除本书外另著有《团队之美》《Head First C#》《Head First PMP》等书。

"Andrew和Jenny对'敏捷'概念给出了质朴、易于理解的定义，引起了人们的共鸣，这就是他们的贡献。通过阅读本书，你可以首先纵览所有敏捷方法，然后再决定使用哪一种，并且能够了解整个敏捷系统及其运作原理。"

——Johanna Rothman
《项目管理之道》作者，
资深敏捷顾问
www.jrothman.com

SOFTWARE DEVELOPMENT/AGILE

封面设计：Ellie Volckhausen 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／敏捷开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-44755-5

ISBN 978-7-115-44755-5
定价：79.00元